Testbarkeitsanalyse von Abhängigkeiten in objektorientierter Software

Diplomarbeit

von Edgar Merl
- April 2003 -

Betreuer:
Prof. Dr. Hans-Werner Six
Lehrgebiet Praktische Informatik III
Fachbereich Informatik
FernUniversität Hagen

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig und nur unter Nutzung der angegebenen Literatur und Hilfsmittel angefertigt zu haben. Wörtlich übernommene Sätze und Satzteile sind als Zitate belegt, andere Anlehnungen hinsichtlich Aussage und Umfang unter den Quellenangaben kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Nürnberg, 11. April 2003

Edgar Merl (Matr.-Nr.: 4181824)

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung der Diplomarbeit in vielfältiger Weise unterstützt haben. Ein besonderes Dankeschön gebührt Stefan Jungmayr. Er hat mir nicht nur ein Verständnis von wissenschaftlicher Arbeitsweise und umfangreiches Fachwissen vermittelt, er stand mir auch jederzeit bei allen Fragen und Problemen geduldig und erschöpfend mit Rat und Tat zur Seite. Ich kann jedem Diplomanden nur wünschen, jemanden wie ihn bei der Erstellung einer Diplomarbeit an der Seite zu wissen. All meinen Freunden danke ich für den seelischen und moralischen Beistand, den sie mir in den letzten Monaten gaben. Ohne ihre Unterstützung hätte ich es sicher nicht geschafft, diese Arbeit zu einem vernünftigen Ende zu bringen. Zusätzlicher Dank gebührt Renate Merl, die mich bis zur letzten Sekunde durch Korrektur lesen unterstützt hat. Silvia Baumann danke ich dafür, dass sie in den letzten Monaten der Ruhepol meines Lebens war. Last but not least geht mein Dank an meine Arbeitskollegen – für ihre Nachsicht, ihre Ratschläge und dafür, dass auch sie stets ein aufmunterndes Wort für mich übrig hatten.

Vielen Dank!

Inhaltsverzeichnis

TEIL I - EINLEITUNG 1

1 ABHÄNGIGKEITEN UND TESTBARKEIT 1
1.1 Testbarkeit – Motivation und Definition 1
1.2 Einfluss von Abhängigkeiten auf Testbarkeit 2
1.3 Testbarkeitsmetriken zur Identifizierung testkritischer Abhängigkeiten 3

2 AUFGABENSTELLUNG 5

3 AUFBAU DER ARBEIT 6

TEIL II - WERKZEUGERSTELLUNG FÜR ABHÄNGIGKEITSANALYSE 7

4 ANFORDERUNGSERMITTLUNG 7
4.1 Präzisierung der Aufgabenstellung 7
4.2 Anwendungsfälle 8
4.3 Zerlegung in Teilaufgaben (T 1 bis T 5) 8

5 ANALYSE 9
5.1 Einleitung 9
5.2 Metrikwerkzeug ImproveT 9
5.3 Entwicklungsumgebung Together 9
5.4 Erweiterung von Together (T 1) 10
5.5 Ermittlung von Abhängigkeiten (T 2) 11
5.6 Erzeugung des Abhängigkeitsgraphen (T 3) 16
5.7 Export des Abhängigkeitsgraphen und Start der Metrikberechnung (T 4) 20
5.8 Anzeige der Testbarkeitsmetriken (T 5) 21

6 ENTWURF 24
6.1 Einleitung 24
6.2 Erweiterung von Together (class Design2Test) 24
6.3 Komponente Abhängigkeitssuche (package search) 25
6.4 Komponente Abhängigkeitsgraph (package data) 28
6.5 Komponente Analysesteuerung (package analysis) 29
6.6 Komponente Metrikauswahl (package selection) 29
6.7 Ausgabekomponenten (package results) 30
6.8 Komponente Modulkoordination (class D2TMediator) 34
6.9 Abschließendes Klassendiagramm des Entwurfs 37

7 IMPLEMENTIERUNG 38
7.1 Einleitung 38
7.2 Erweiterung von Together (class Design2Test) 38
7.3 Komponente Abhängigkeitssuche (package search) 38
7.4 Komponente Abhängigkeitsgraph (package data) 41
7.5 Komponente Analysesteuerung (package analysis) 42
7.6 Komponente Metrikauswahl (package selection) 43
7.7 Ausgabekomponenten (package results) 43
7.8 Komponente Modulkoordination (class D2TMediator) 44
7.9 Probleme der Implementierungsphase 45
7.10 Fazit 46

TEIL III - FALLBEISPIEL 47

8 EINFÜHRUNG 47
8.1 Motivation 47
8.2 Beschreibung des zu analysierenden Softwaresystems 49

9 TESTBARKEITSANALYSE AUSGEWÄHLTER ABHÄNGIGKEITEN 50
9.1 Einleitung 50
9.2 Top rACD-Abhängigkeiten 52
9.3 Adaption der Testbarkeitsmetriken 84

10 ENTWURFSPROBLEME UND TESTBARKEIT 85
10.1 Einleitung 85
10.2 Entwurfsüberblick SeminarIS 86
10.3 Verletzung der Drei-Schicht-Architektur 90
10.4 Zugriff auf Kontrollschicht durch Basiskomponenten 93
10.5 Zugriff auf Datenhaltung durch Basiskomponente 94
10.6 Zugriff auf globale Basiskomponenten 96
10.7 Realisierung der Domänenklassen als Fassaden 98
10.8 Erzeugung der Schnittstellenklassen 102

11 ENTFERNUNG TESTKRITISCHER ABHÄNGIGKEITEN DURCH REFAKTORISIERUNG 104
11.1 Einleitung 104
11.2 Realisierung einer strikten Drei-Schichten-Architektur 104
11.3 Kein externer Zugriff auf Kontrollschicht 106
11.4 Kein externer Zugriff auf Datenhaltung 108
11.5 Veränderter Zugriff auf globale Basiskomponenten 109
11.6 Direkter Zugriff auf Ordner und Relationen 115
11.7 Einführung einer Schnittstellenklassen-Factory 119

TEIL IV - RESÜMEE 126

12 ZUSAMMENFASSUNG 126

13 AUSBLICK 129

LITERATURVERZEICHNIS 130

ANHANG A ANWENDUNGSFÄLLE TESTBARKEITSANALYSE 132

ANHANG B ÜBERSICHT JAVA-ABHÄNGIGKEITEN 140

ANHANG C BENUTZERHANDBUCH DESIGN2TEST 163

Teil I - Einleitung

Im ersten Teil dieser Arbeit wird der Zusammenhang von Testbarkeit eines Softwaresystems und den im System vorhandenen Abhängigkeiten beschrieben und ein Weg gezeigt, wie der Einfluss von Abhängigkeiten auf die Testbarkeit mit Hilfe von Metriken eingeschätzt werden kann. Hierzu werden die Begriffe Testbarkeit und Abhängigkeiten definiert und darauf aufbauend ein Konzept zur Identifizierung testkritischer Abhängigkeiten beschrieben. Ein Überblick über die Aufgabenstellung und über den Aufbau der Arbeit schließt diesen Teil ab.

1 Abhängigkeiten und Testbarkeit

1.1 Testbarkeit – Motivation und Definition

Die Aussage, Testen sei ein wichtiger Qualitätsfaktor bei der Entwicklung von Softwaresystemen, trifft bei jedem Entwickler auf volle Zustimmung. Betrachtet man die umfangreich vorhandene Testliteratur, so lässt sich auch nicht sagen, dass für die Planung und Durchführung von Tests nur wenig Unterstützung zu finden wäre. Und trotzdem ist in vielen Projekten der Testprozess der am meisten vernachlässigte Teil im Rahmen der Softwareentwicklung.

Es gibt sicherlich viele Ursachen hierfür. Kommt der übliche Stress im Programmieralltag auf, so verbleibt in der Regel am wenigsten Zeit für die zumeist letzte Phase im Entwicklungsprozess: Die Testphase. Ist man zudem der Meinung, dass Testen unter dem Strich stets zu Mehraufwand und längeren Entwurfszeiten führt, so steigert dies nicht gerade die Motivation für Tests. Testen wird von vielen Entwicklern ferner als monoton, wenig kreativ und langweilig empfunden. Und außerdem kann man zu guter Letzt immer noch auf die Arbeit der Testabteilung im Hintergrund vertrauen.

Möglicherweise liegen die Ursachen aber auch an anderer Stelle. Neben Faktoren wie

- Güte der Testmethodik,
- Umfang der Unterstützung durch Testwerkzeuge,
- Möglichkeit zur Bereitstellung einer adäquaten Testinfrastruktur,
- Ausbildung und Erfahrung der Tester,
- Korrektheit und Vollständigkeit der Spezifikation, gegen die getestet werden soll,

haben insbesondere die

- Eigenschaften der zu testenden Software

maßgeblichen Einfluss auf die Komplexität des Tests und den zu leistenden Testaufwand. Der Grad, zu dem die zu testende Software die Durchführung der Testaufgaben zur Überprüfung von zu ermittelnden Testkriterien erleichtert, wird als Testbarkeit bezeichnet **[**IEEE90**]**.

Die Abneigung gegen die systematische und umfassende Durchführung von Tests wird also auch darin begründet sein, dass Softwaresysteme, die ohne Rücksicht auf ihre Testbarkeit entworfen wurden,

Eigenschaften besitzen, die in der Tat zu einem äußerst unangenehmen und aufwändigen Testprozess führen. Eine mögliche Lösung dieses Problems ist unter anderem auch Bestandteil des Konzepts von *Extreme Programming* (XP) **[**Beck00**]**. Mit dem Aufkommen und der Propagierung von XP fand die Durchführung von Komponententests (*Unit Tests*) und insbesondere deren Erstellung vor oder parallel zum eigentlichen Entwicklungscode (*Test First Design*) breites Interesse.

Während also beim Großteil der in der Literatur beschriebenen klassischen Entwurfs- und Testverfahren die Testaktivitäten und Testfälle auf Basis eines detaillierten, vorab bereits fertig gestellten Entwurfs definiert werden, erfolgt dies in XP parallel zu Softwaredesign und Implementierung. Das auf diesem Wege entstehende Design scheint prinzipiell besser für die Erledigung der Testaufgaben geeignet zu sein.

Daraus folgt das Bestreben, unabhängig vom verwendeten Vorgehensmodell dem Entwickler bereits so früh als möglich eine Rückmeldung über die Testbarkeit seines Designs zu geben. Hierzu soll ihm mit der vorliegenden Arbeit ein Werkzeug an die Hand gegeben werden, das ihn auch bei klassischen Entwurfsverfahren in die Lage versetzt, frühzeitig Hinweise auf jene Eigenschaften und Strukturen seiner Software zu erhalten, die negativen Einfluss auf die Testbarkeit seines Projekts ausüben.

1.2 Einfluss von Abhängigkeiten auf Testbarkeit

Einer der Faktoren, der Auswirkungen auf die Testbarkeit eines Programms besitzt, sind Abhängigkeiten zwischen Programmteilen. Benötigt eine Komponente (hierbei kann es sich beispielsweise um eine Klasse, Interface oder Paket handeln) zur Erfüllung ihrer Aufgaben keine weitere Funktionalität aus anderen Komponenten, vereinfacht dies ihre Betrachtung in der Testphase. Komplexere Anwendungsfunktionalität lässt sich jedoch erst dadurch realisieren, dass verschiedene Komponenten miteinander kommunizieren. Dieses Zusammenwirken führt zu unvermeidbaren Abhängigkeiten unter diesen Komponenten.

Mit Abhängigkeiten sind im Rahmen dieser Arbeit syntaktische Abhängigkeiten von Programmkomponenten gemeint. Der Begriff der syntaktischen Abhängigkeit einer Komponente A von Komponente B bedeutet, dass Komponente A ohne das Vorhandensein von Komponente B nicht kompilierfähig ist. Daneben existieren noch semantische Abhängigkeiten von Programmkomponenten, wenn beispielsweise eine Komponente in ihrer Implementierung auf ein bestimmtes Verhalten einer anderen Komponente vertraut. Die Einschränkung auf syntaktische Abhängigkeiten im Rahmen dieser Arbeit basiert auf dem Umstand, dass syntaktische Abhängigkeiten maschinell erfasst werden können. Semantische Abhängigkeiten sind einer automatisierten Erkennung kaum oder nur mit wesentlich höherem Aufwand zugänglich.

Durch Abhängigkeiten wird eine transitive Relation zwischen den Komponenten eines Systems definiert. Dies bedeutet zum einen, dass jede Abhängigkeit eine Richtung besitzt, und zwar von der abhängigen Komponente (*dependent* oder *client class*) zu derjenigen Komponente, die im Rahmen der Abhängigkeit Funktionalität liefert (*dependee* oder *server class*). Somit können zwischen zwei Komponenten maximal zwei Abhängigkeiten (mit jeweils entgegengesetzter Richtung) existieren. Zudem führen direkte Abhängigkeiten der Komponente A von Komponente B und der Komponente B von Komponente C zu einer indirekten Abhängigkeit der Komponente A von Komponente C.

In einem der Anhänge (Anhang B - *Übersicht Java-Abhängigkeiten*) erfolgt eine Aufstellung und Klassifizierung der möglichen syntaktischen Abhängigkeiten in Java-Programmen. Bezüglich der Möglichkeit zur isolierten Testdurchführung lassen sich die ermittelten Abhängigkeiten in zwei Gruppen einteilen. Während die Abhängigkeiten der einen Gruppe durch ihre Eigenschaften den isolierten Test einer Komponente begünstigen, verhindern oder zumindest erschweren dies die Abhängigkeiten der anderen Gruppe.

Die Schwierigkeit, eine Komponente in der Testphase aufgrund von Abhängigkeiten nicht für sich alleine betrachten zu können, führt erfahrungsgemäß zu diversen Testproblemen. Je größer die Zahl der zu berücksichtigenden Abhängigkeiten einer Komponente, desto wahrscheinlicher wird ein hoher Aufwand für die Testfallerstellung und die Durchführung der Tests, desto schwieriger im Regelfall die Fehlererkennung und umso weniger Flexibilität besteht bei der Bestimmung einer geeigneten Testreihenfolge.

Ein besonderes Testproblem stellen ferner zyklische Abhängigkeiten dar. Grundsätzlich können alle an einem Zyklus beteiligten Komponenten nur gemeinsam getestet werden. Dies verschärft die oben bereits angeführten Probleme wie beispielsweise die Lokalisierung von Fehlern. Durch den Einsatz von Stellvertretern (*Stubs*) können die zyklischen Abhängigkeiten zwar aufgelöst werden, dies erfordert jedoch zusätzlichen Aufwand und führt zu erhöhter Komplexität bei der Testfallerstellung.

1.3 Testbarkeitsmetriken zur Identifizierung testkritischer Abhängigkeiten

In der Literatur existieren bereits verschiedene Vorschläge für Metriken zur Beurteilung der Testbarkeit eines Systems. Darunter finden sich auch bereits Ansätze, die sich mit der Architektur und den Abhängigkeiten innerhalb eines Systems auseinandersetzen **[**LoSh98**]**, **[**ISO9126**]** und **[**Lako96**]**. Diese beurteilen zwar anhand der Abhängigkeiten die Testbarkeit einzelner Komponenten oder des gesamten Systems, sie liefern jedoch keine Hinweise darauf, welche konkreten Abhängigkeiten Auslöser für einen Mangel an Testbarkeit sind.

Um in einem Softwaresystem jene Abhängigkeiten identifizieren zu können, die in besonderem Maße die Testbarkeit beeinträchtigen, wurde in **[**Jung02**]** ein neuer Ansatz vorgeschlagen. Er basiert auf der Erwartung, dass die Hauptziele im Testprozess, nämlich mit möglichst geringem Aufwand und in möglichst kurzer Zeit eine möglichst hohe Qualität der zu testenden Software zu erreichen, insbesondere bei möglichst hoher Unabhängigkeit der zu testenden Komponenten erreicht werden können. Für die Einschätzung dieser Unabhängigkeit werden in einem ersten Schritt Metriken zur Messung bestimmter Eigenschaften des Systems definiert. Dabei handelt es sich beispielsweise um folgende Metriken:

- **ACD** – **A**verage **C**omponent **D**ependency:
  Hierbei handelt es sich um die durchschnittliche Anzahl von Komponenten, von denen eine Komponente des Systems direkt oder indirekt abhängt.
- **NFD** – **N**umber Of **F**eedback **D**ependencies
  Diese Metrik gibt die Anzahl der Abhängigkeiten an, deren Entfernung zur Auflösung von Zyklen des Systems führt.
- **NSBC** – **N**umber Of **S**tubs Needed To **B**reak **C**ycles
  Bei der Testdurchführung ist das Durchbrechen von zyklischen Abhängigkeiten zwischen Anwendungskomponenten durch den Einsatz von Stellvertretern (Stubs) möglich. Diese Metrik liefert die Anzahl der Stellvertreter, die zum Durchbrechen aller Zyklen benötigt werden.
- **NCDC** – **N**umber Of **C**omponents Within **D**ependency **C**ycles
  Verzichtet man auf den Einsatz von Teststellvertretern, so können die Komponenten eines Zyklus nur zusammenhängend getestet werden. Diese Metrik liefert die Anzahl der Komponenten, die in Abhängigkeitszyklen involviert sind.

Im zweiten Schritt wird für sämtliche Abhängigkeiten des Systems eine Kenngröße definiert, die erkennen lässt, welchen Einfluss eine einzelne Abhängigkeit auf eine der gemessenen Systemeigenschaften hat. Dies erfolgt über die Einführung der Reduktionsmetriken (*reduction metrics*) für Abhängigkeiten. Eine Reduktionsmetrik bezieht sich dabei stets auf eine der im ersten Schritt definierten Projektmetriken. Der Wert einer Reduktionsmetrik gibt an, um welchen

prozentualen Anteil sich die für das Gesamtsystem ermittelte Metrik verändern würde, falls die betreffende Abhängigkeit aus dem System entfernt werden könnte. Besitzt das Gesamtsystem beispielsweise einen ACD-Wert von 90 und würde die Entfernung der Abhängigkeit zu einem ACD-Wert von 81 führen, so beträgt der Wert der Reduktionsmetrik (rACD) für diese Abhängigkeit 10 Prozent.

Als testkritisch werden schließlich jene Abhängigkeiten betrachtet, die vergleichsweise hohe Werte von Reduktionsmetriken aufweisen. Diese testkritischen Abhängigkeiten sind potentielle Kandidaten, mit deren Beseitigung bessere Voraussetzungen zur Erreichung der eingangs erwähnten Hauptziele im Testprozess erreicht werden sollten. Mit dieser Fragestellung wird sich die vorliegende Arbeit in Teil III - *Fallbeispiel* - näher beschäftigen.

2 Aufgabenstellung

Das erste Teilziel der vorliegenden Arbeit ist es, Softwareentwicklern künftig ein Werkzeug zur Testbarkeitsanalyse als Bestandteil einer Entwicklungsumgebung anzubieten. Die Testbarkeitsanalyse soll insbesondere Hinweise auf mögliche testkritische Abhängigkeiten zwischen einzelnen Komponenten liefern, da diese einen potentiell hohen Einfluss auf die Testbarkeit besitzen (siehe Abschnitt 1.3). Ihre Untersuchung scheint deshalb besonders lohnenswert, da ihre Entfernung eine vergleichsweise hohe Verbesserung der Testbarkeit verspricht.

Zur Berechnung der in Abschnitt 1.3 eingeführten Metriken wurde bereits ein Metrikwerkzeug implementiert, das für eine Integration in eine Entwicklungsumgebung genutzt werden kann. Bei der Erweiterung der Entwicklungsumgebung um die Funktionalität zur Testbarkeitsanalyse sind im Wesentlichen folgende Aufgaben zu erfüllen:

- Ermittlung aller in einem gegebenen Softwaresystem vorhandenen Abhängigkeiten,
- Export der Abhängigkeiten in das vorhandene Metrikwerkzeug zur Berechnung der Testbarkeitsmetriken,
- Darstellung der ermittelten Abhängigkeiten und der für sie berechneten Metriken innerhalb der Entwicklungsumgebung.

Die schrittweise Verfeinerung und Ausarbeitung der Aufgabenstellung erfolgt zu Beginn des Entwicklungsteils dieser Arbeit (Kapitel 4).

Das zweite Teilziel dieser Arbeit sieht die Verwendung des entwickelten Werkzeugs vor. Mit seiner Hilfe soll für ein existierendes Softwaresystem der tatsächliche Einfluss testkritischer Abhängigkeiten untersucht und möglichst exakt bestimmt werden. Hierbei sind im Wesentlichen folgende Aufgaben zu erfüllen:

- Bewertung der als testkritisch identifizierten Abhängigkeiten hinsichtlich ihrer Entstehung und Interpretation der Metriken,
- Erarbeiten von Lösungsvorschlägen zur Beseitigung der Abhängigkeiten, beispielhafte Entfernung der Abhängigkeiten und Einschätzung des hierfür zu leistenden Aufwandes,
- Bestimmung und Vergleich des zu leistenden Testaufwandes vor und nach der Beseitigung testkritischer Abhängigkeiten.

Die schrittweise Verfeinerung und Ausarbeitung der Aufgabenstellung erfolgt zu Beginn des Analyseteils dieser Arbeit (Kapitel 8).

3 Aufbau der Arbeit

Teil II - *Werkzeugerstellung für Abhängigkeitsanalyse* – dieser Arbeit beschreibt den bei der Werkzeugerstellung durchlaufenen Entwicklungsprozess, beginnend mit der Anforderungsermittlung in Kapitel 4. Ergänzend zur Verfeinerung und Präzisierung der Aufgabenstellung aus Kapitel 2 werden *UML*-Anwendungsfälle verwendet, um die funktionalen Anforderungen der zu realisierenden Software festzuhalten. Um die Komplexität in der Entwurfsphase zu reduzieren, erfolgt die Zerlegung der Gesamtaufgabe in verschiedene Teilaufgaben.

Da die Realisierung der Teilaufgaben ganz entscheidend von den Eigenschaften und Schnittstellen der beiden beteiligten Systeme bestimmt ist, erfolgt in Kapitel 5 zunächst eine Analyse der zu integrierenden Systeme und ihrer Schnittstellen. Im Mittelpunkt des Interesses steht hierbei die von den Systemen gebotene Unterstützung bei der Realisierung der Teilaufgaben.

Ausgehend von den identifizierten Teilaufgaben und auf Basis der vorgeschalteten Analyse erfolgt in Kapitel 6 der Entwurf der einzelnen Teilkomponenten. Es werden Komponenten für die Suche nach Abhängigkeiten und die Erstellung eines Abhängigkeitsgraphen entworfen, eine Schnittstellenkomponente zur Steuerung der Metrikberechnung, eine Komponente zur Auswahl der zu berechnenden Metriken und mehrere Ausgabekomponenten. Zur Steuerung des Zusammenspiels aller Komponenten wird eine Koordinationskomponente eingeführt.

Aufbauend auf den Entwurf erfolgt die Beschreibung ausgewählter Teile der Implementierung in Kapitel 7. Dem erforderlichen Implementierungsaufwand entsprechend, nimmt die Realisierung der Suchkomponente hier den größten Raum ein. Den Abschluss des Kapitels bildet eine Beschreibung verschiedener Probleme bei der Implementierung.

Teil III - *Fallbeispiel* - beschreibt die Verwendung des entwickelten Werkzeugs bei der Testbarkeitsanalyse der Abhängigkeiten eines gegebenen Softwaresystems. Zunächst wird in Kapitel 8 für diesen Teil der Arbeit wiederum die Aufgabenstellung aus Kapitel 2 präzisiert und das zu analysierende Softwareprojekt vorgestellt.

Begonnen wird die Testbarkeitsanalyse in Kapitel 9 mit der Ermittlung der Testbarkeitsmetriken des zu untersuchenden Softwaresystems. Auf Basis der erhaltenen Ergebnisse werden 30 testkritische Abhängigkeiten für eine nähere Untersuchung ausgewählt. Im Rahmen der Untersuchung erfolgt eine Beschreibung der Ursachen der Abhängigkeit und eine Bewertung der Abhängigkeit unter dem Gesichtspunkt der Testbarkeit.

In Kapitel 10 erfolgt eine Untersuchung der Entwurfsprobleme, die bei der Analyse in Kapitel 9 als Hauptursachen für Testprobleme ermittelt wurden. Es werden verschiedene Entwurfsprobleme, wie die Verletzung von Hierarchien in Schichtenstrukturen, der Zugriff auf globale Basiskomponenten oder die Fassadenfunktion der Domänenklassen und ihre jeweiligen Auswirkungen auf die Testbarkeit im Detail beschrieben.

Nach der Untersuchung und Beschreibung der Entwurfsprobleme werden in Kapitel 11 Ansätze für die Beseitigung der Entwurfsprobleme durch eine Veränderung der Implementierung beschrieben. Auf Basis dieser Ansätze erfolgt die Refaktorisierung des untersuchten Softwaresystems und die Bestimmung der hierfür erforderlichen Aufwände.

Den Abschluss der Arbeit bildet Teil IV - *Resümee*. Hier werden in Kapitel 12 noch einmal die wichtigsten Ergebnisse dieser Arbeit zusammengefasst und in Kapitel 13 ein Ausblick auf eine mögliche Fortführung der Arbeit zu diesem Thema gegeben.

Teil II - Werkzeugerstellung für Abhängigkeitsanalyse

Im zweiten Teil dieser Arbeit wird die Entwicklung des integrierten Werkzeugs zur Abhängigkeitsanalyse beschrieben. Die Beschreibung orientiert sich an den verschiedenen Stufen des Entwicklungsprozesses und umfasst jeweils ein Kapitel für die Anforderungsermittlung, die Analyse, den Entwurf und die Implementierung der zu entwickelnden Softwarekomponente.

4 Anforderungsermittlung

4.1 Präzisierung der Aufgabenstellung

Erstes Ziel der vorliegenden Arbeit ist, den Nutzern einer integrierten Entwicklungsumgebung innerhalb dieser Entwicklungsumgebung Funktionalität zur Identifizierung testkritischer Bereiche in ihrem System zur Verfügung zu stellen (siehe Aufgabenstellung in Kapitel 2).

Zur Umsetzung des in Abschnitt 1.3 vorgestellten Konzeptes zur Identifizierung von testkritischen Abhängigkeiten existierte bereits ein von Stefan Jungmayr implementiertes Metrikwerkzeug mit dem Namen *ImproveT*. Als Entwicklungsumgebung, die um die Funktionalität von *ImproveT* erweitert werden sollte, wurde das *Together ControlCenter* (kurz: *Together*) der zwischenzeitlich von der Firma Borland übernommenen Firma Togethersoft ausgewählt. Eine umfassende Beschreibung von *ImproveT* und *Together* erfolgt im Rahmen der Analyse der zu integrierenden Werkzeuge und ihrer Schnittstellen in Kapitel 5.

Zur Erfüllung der ersten Teilaufgabe dieser Diplomarbeit war die Integration von *ImproveT* in *Together* vorgesehen. Im Rahmen der Integration sollte im Einzelnen folgende Funktionalität realisiert werden:

- Anstoßen der Testbarkeitsanalyse über die Menüstruktur von *Together* mit der Möglichkeit zur Auswahl der zu berechnenden Testbarkeitsmetriken.
- Analyse der in *Together* verwalteten Modellelemente (Diagramme, Sourcecode) in bezug auf Abhängigkeiten zwischen einzelnen Programmkomponenten.
- Ermittlung dieser Abhängigkeiten und Export nach *ImproveT*.
- Starten der Abhängigkeitsanalyse durch *ImproveT*.
- Darstellung der von *ImproveT* berechneten Testbarkeitsmetriken für die im Abhängigkeitsgraphen gespeicherten Abhängigkeiten und Programmkomponenten in Listenform mit Sortiermöglichkeit.
- Graphische Darstellung der Abhängigkeiten und der beteiligten Programmkomponenten mit der Möglichkeit zur Kennzeichnung der Graphelemente nach Ihrem Einfluss auf die Testbarkeit des Systems.
- Möglichkeit der wechselseitigen Navigation zwischen der Listendarstellung von Abhängigkeiten und Komponenten, deren graphischer Darstellung und den in

Together vorhandenen Modellelementen sowie dem Sourcecode der analysierten Software über beispielsweise Kontextmenüs.

- Gegebenenfalls Möglichkeit zum Wechsel zwischen Klassen-Ebene und Paket-Ebene bei der Metrikberechnung. Die Realisierung dieser Anforderung wurde im Verlauf der Arbeit allerdings auf einen späteren Zeitpunkt verschoben (siehe auch Abschnitt 6.8).

4.2 Anwendungsfälle

Zur exakten Beschreibung der bei der Erweiterung von *Together* zu realisierenden funktionalen Anforderungen wurden Anwendungsfälle verwendet. Die Notation der Anwendungsfälle erfolgte auf Basis der *Unified Modelling Language (UML)* [OUML00]. Ausgehend von der Aufgabenstellung wurden die daraus resultierenden Funktionalitäten in einem Anwendungsfalldiagramm und in textuellen Anwendungsfallspezifikationen beschrieben. Dabei wurden einige der obigen Anforderungen präzisiert und teilweise auch um zusätzliche Funktionalitäten erweitert. Die vollständige Beschreibung der Anwendungsfälle findet sich in Anhang A zu dieser Arbeit.

4.3 Zerlegung in Teilaufgaben (T 1 bis T 5)

Für die zu leistende Integration von *ImproveT* in *Together* lässt sich auf Basis der Aufgabenstellung und der beschriebenen Anwendungsfälle für eine erste Analyse der beteiligten Systeme folgende Zerlegung in Teilaufgaben vornehmen:

T 1. Integration der zu implementierenden Erweiterung in die Menüstruktur von *Together,* Anbindung der Erweiterung an das *Together Open API* und Realisierung als *Together*-Modul.

T 2. Realisierung eines Teilmoduls auf Basis des *Together Open API* zur Untersuchung der Modellelemente und des Sourcecodes nach Abhängigkeiten zwischen Programmkomponenten.

T 3. Aufbau eines Abhängigkeitsgraphen nach der von *ImproveT* vorgegebenen Schnittstellenspezifikation auf Grundlage der ermittelten Abhängigkeiten.

T 4. Export des Abhängigkeitsgraphen nach *ImproveT*, Möglichkeit zur Auswahl der zu berücksichtigenden Metriken durch den Anwender und Starten der Metrikberechnung in *ImproveT*.

T 5. Anzeige der ermittelten Testbarkeitsmetriken in *Together*, graphische Darstellung der Abhängigkeiten, wechselseitige Navigationsmöglichkeiten in den Anzeigekomponenten.

Diese Teilaufgaben bilden die Basis für die im nächsten Kapitel durchgeführte Analyse. Diese Analyse bereitet den Entwurf und die Implementierung der *Together*-Erweiterung in Kapitel 6 und 7 vor. Zentrales Thema der Analyse wird die Beschreibung der für die Teilaufgaben jeweils benötigten Schnittstellen von *ImproveT* und *Together* sein.

5 Analyse

5.1 Einleitung

In diesem Kapitel werden die Vorarbeiten für den Entwurf und die Implementierung der *Together*-Erweiterung um *ImproveT* geleistet. Nach einer kurzen einführenden Charakterisierung der beiden an der Integration beteiligten Softwarekomponenten, erfolgt die Analyse dieser Komponenten im Hinblick auf die im Rahmen der Integration zu leistenden Teilaufgaben aus Abschnitt 4.3. Bei der Durchführung der Analyse steht die Beschreibung der von den beteiligten Komponenten zur Verfügung gestellten Schnittstellen im Mittelpunkt.

5.2 Metrikwerkzeug ImproveT

Zur Umsetzung des in Abschnitt 1.3 vorgestellten Konzeptes zur Identifizierung von testkritischen Abhängigkeiten wurde von Stefan Jungmayr ein Metrikwerkzeug mit dem Namen *ImproveT* realisiert. Dieses Werkzeug analysiert Abhängigkeiten objektorientierter Software, die in einem mehrschichtigen Abhängigkeitsgraphen vorliegen. Auf Basis dieses Abhängigkeitsgraphen werden die Testbarkeitsmetriken, Reduktionsmetriken und zusätzliche Informationen für die Abhängigkeiten und Programmkomponenten eines Softwaresystems errechnet. Ferner bietet *ImproveT* die Möglichkeit zur Visualisierung der Komponenten des Systems und der zwischen ihnen ermittelten Abhängigkeiten als Graph.

Die Implementierung von *ImproveT* ist vollständig in Java gehalten. Parallel zu dieser Diplomarbeit und der Integration von *ImproveT* in *Together* (siehe Abschnitt 5.3) erfolgte durch Stefan Jungmayr die Fortentwicklung des Werkzeugs (auch auf Basis der in dieser Arbeit gewonnenen Erkenntnisse). Die Beschreibung des Werkzeugs basiert auf dem gegen Ende der Diplomarbeit verfügbaren Release.

Die Berechnung der Testbarkeitsmetriken durch *ImproveT* läuft im Wesentlichen wie folgt ab:

- Erzeugung eines Abhängigkeitsgraphen über das Einfügen von Abhängigkeiten und Programmkomponenten des zu analysierenden Systems als Kanten und Knoten des Graphen,
- Analyse des Abhängigkeitsgraphen und darauf basierend Berechnung der Abhängigkeits- und Komponentenmetriken,
- Bereitstellung der Analyseergebnisse für die weitere Verarbeitung.

Eine nähere Beschreibung der Funktionalität und der angebotenen Schnittstellen von *ImproveT* erfolgt bei der Analyse der Teilaufgaben in den nachfolgenden Abschnitten dieses Kapitels.

5.3 Entwicklungsumgebung Together

Im Lehrgebiet Praktische Informatik III der FernUniversität Hagen wurden unter anderem im Rahmen von Software-Praktika gute Erfahrungen mit der Software-Entwicklungsplattform *Together ControlCenter* (kurz: *Together*) der Firma *TogetherSoft* gemacht. Diese Entwicklungsumgebung unterstützt den kompletten Software-Entwicklungsprozess: Neben der Implementierung werden auch die Analyse- und Entwurfsphase durch umfangreiche Funktionalität unterstützt.

Together stellt als grundlegende Voraussetzung für die Integration von *ImproveT* eine frei zugängliche Java-Programmierschnittstelle standardmäßig zur Verfügung. Diese als *Together Open API* bezeichnete Programmierschnittstelle bildet die Basis für die Entwicklung von Together-Erweiterungen. Sie bietet hierzu die Möglichkeit, auf alle Modellelemente sowie den kompletten Sourcecode eines Softwareprojekts zugreifen zu können. Ein auf Grundlage dieser Schnittstelle entwickeltes Modul zur Bestimmung der Abhängigkeiten eines Systems ist ebenfalls im Lieferumfang enthalten. Leider zeigte sich bald, dass die Funktionalität dieses Moduls für die Wiederverwendung bei der Realisierung der Abhängigkeitssuche (Teilaufgabe T 2) nicht ausreichte.

Together zeichnet sich ferner dadurch aus, dass alle gespeicherten Informationen automatisch synchronisiert werden [ToSZ02]. Das bedeutet insbesondere, dass zwischen allen Modellelementen des Entwurfs, wie z.B. Klassendiagrammen, und dem implementiertem Sourcecode zu jeder Zeit vollständige Konsistenz herrscht. Durch die dadurch vorhandene Abbildung von beispielsweise Klassendiagrammen im Sourcecode, wird die notwendige Ermittlung der vorhandenen Abhängigkeiten innerhalb eines Softwareprojekts vereinfacht. Dies erleichtert es auch, bereits mit Beginn der Entwurfsphase Testbarkeitsanalysen vornehmen zu können.

Eine nähere Beschreibung des Aufbaus und der Funktionalität des *Together Open API* erfolgt bei der Analyse der Teilaufgaben in den nachfolgenden Abschnitten dieses Kapitels.

5.4 Erweiterung von *Together* (T 1)

5.4.1 Aufbau des Together Open API

Das *Together Open API* gliedert sich auf oberster Ebene in drei Pakete. Diese Pakete realisieren verschiedene Zugriffsebenen auf die Entwicklungsplattform und dem der Plattform zugrunde liegenden internen Programmmodell. Die Struktur der Programmierschnittstelle ist dreischichtig konzipiert:

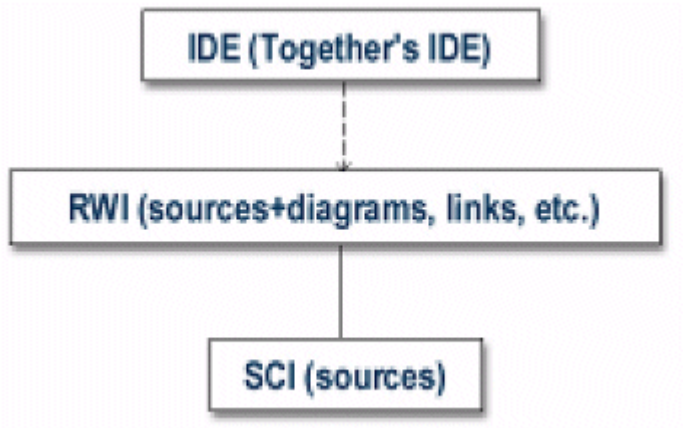


Abbildung 1: Die drei Schichten des Together Open API (Quelle:[ToUG02])

Die drei Schichten des *Together Open API* ermöglichen den Zugriff auf die verfügbaren Informationen und Programmkomponenten eines in *Together* verwalteten Softwaresystems (*Together*-Projekt). In den tieferen Schichten ist auch der ändernde Zugriff auf das Modell möglich:

- **IDE**
  Diese Schicht stellt die Ausgabeschnittstelle dar. Sie bietet den Zugriff auf alle Ausgabekomponenten der Entwicklungsumgebung. Hierüber lässt sich die Interaktion mit dem Benutzer und die Präsentation von Informationen steuern. Diese Schicht enthält keine Zugriffsmethoden auf die Modellelemente oder den Programmcode eines *Together*-Projekts.
- **RWI**
  Diese Schicht bietet den Zugriff auf alle in einem *Together*-Projekt enthaltenen Modellelemente. Bei den Modellelementen handelt es sich beispielsweise um die in Diagrammen angezeigten Pakete, Klassen, Attribute und Methoden. Auch der Zugriff auf die darstellenden *UML*-Elemente, wie Klassen- oder Interaktionsdiagramme, ist in dieser Schicht möglich. Die Informationen der Modellelemente können abgerufen aber auch adaptiert werden. Ferner können dem Modell neue Elemente hinzugefügt werden.
- **SCI**
  Bei dieser Schicht handelt es sich um die am feinsten granulierte Ebene der internen Repräsentation eines Projekts durch *Together*. Über diesen Weg ist der lesende und schreibende Zugriff auf den Sourcecode eines *Together*-Projekts möglich. Durch das Konzept der permanenten Synchronisation von Programmcode und Modellierung (siehe Abschnitt 5.3) wird über diese Schicht auch die Modellierung beeinflusst. Für die Zwecke dieser Arbeit ist insbesondere von Bedeutung, dass auf dieser Ebene der Sourcecode des kompletten Projekts durchlaufen und nach Abhängigkeiten zwischen einzelnen Programmkomponenten analysiert werden kann.

5.4.2 Integration von Modulen über das Open API

Eine Erweiterung von *Together* auf der Basis seines *Open API* ist über sogenannte *Together*-Module möglich. Derartige Module können als Javaklassen realisiert werden, die über die vorstehend beschriebenen verschiedenen Schichten des *Together Open API* auf die Informationen und Elemente des *Together*-Modells lesend und schreibend zugreifen. Ein Überblick zur Erweiterung von *Together* über Module findet sich in **[**ToUG02**]**.

5.5 Ermittlung von Abhängigkeiten (T 2)

Nach der Aufgabenstellung sollen alle Modellelemente und der Java-Sourcecode eines in *Together* als Projekt verwalteten objektorientierten Softwaresystems nach dem Auftreten von Abhängigkeiten durchsucht werden. Daraus ergaben sich zunächst folgende Fragen:

- Welche Abhängigkeiten zwischen welchen Programmkomponenten können in einem objektorientierten Java-Programm überhaupt entstehen?
- In welchen Schichten des *Together Open API* (siehe Abschnitt 5.4.1) sind Informationen und Daten vorhanden, die bei der Suche nach Abhängigkeiten in einem Projekt berücksichtigt werden müssen?
- In welchen Elementen der zu untersuchenden Schichten können Abhängigkeiten zwischen Programmkomponenten auftreten und wie kann der Zugriff über das *Together Open API* auf diese Elemente erfolgen?

Die Antworten auf diese Fragen werden in den nächsten Abschnitten beschrieben.

5.5.1 Übersicht möglicher Abhängigkeiten in Java-Programmen

Die wichtigsten Entscheidungen, die bei der Bestimmung der im Rahmen der Testbarkeitsanalyse zu berücksichtigenden Abhängigkeiten zu treffen waren, sind nachstehend aufgelistet:

- Alle Abhängigkeiten innerhalb von inneren und anonymen Klassen werden analog zu den Abhängigkeiten der sie umschließenden Klassen berücksichtigt. Dies umfasst beispielsweise auch Vererbung und Implementierung eines Schnittstellentyps bei der Definition von anonymen Klassen.
- In den Blöcken von statischen und Instanz-Initialisierern können ebenfalls Abhängigkeiten auftreten. Diese finden in geeigneter Weise Berücksichtigung.
- Alle Abhängigkeiten zu Java-Basistypen (int, boolean, ...) bleiben unberücksichtigt. Im Rahmen der Testbarkeit sind diese Typen ohne Einfluss, da sie zum Sprachumfang von Java gehören und somit während der Kompilierung in jedem Falle verfügbar sind. *Together* liefert in seinem Modell auch ausschließlich Referenzen auf Klassentypen wie beispielsweise `String` oder auch auf die Wrapperklassen für primitive Datentypen (`Integer`, `Boolean`, ...).
- Abhängigkeiten zwischen Attributen und Methoden innerhalb derselben Klasse werden berücksichtigt. Damit wird es prinzipiell möglich, bei Vorliegen eines Zyklus zwischen zwei Klassen zu entscheiden, ob der Test der Methoden durch eine geeignete serialisierbare Testreihenfolge ohne Implementierung von Teststellvertretern durchgeführt werden kann **[**Wint00**]**. Die Informationen über diese Abhängigkeiten werden derzeit noch nicht ausgewertet.
- *Together* bietet dem Anwender ferner die Möglichkeit, in Klassendiagrammen manuelle Abhängigkeiten zwischen Programmkomponenten zu definieren. Diese Abhängigkeiten werden bei der Analyse ebenfalls erkannt.

Eine vollständige Übersicht über alle in einem Java-Programm möglichen Abhängigkeiten findet sich in Anhang B. Diese Beschreibung ist bereits um die Modellierungsaspekte der Abhängigkeiten in dem für den Export nach *ImproveT* zu erzeugenden Abhängigkeitsgraphen erweitert (siehe auch Abschnitt 5.6.3).

5.5.2 Lokalisierung der Abhängigkeiten in *Together*

Nach der Aufgabenstellung müssen bei der Feststellung der Abhängigkeiten sowohl Abhängigkeiten aufgrund der in *Together* möglichen Entwurfsmodellierung als auch der darauf aufbauenden Implementierung des Programms erkannt werden. Wie bereits bei der einleitenden Charakterisierung von *Together* beschrieben (siehe Abschnitt 5.3), besitzt *Together* die in diesem Zusammenhang vorteilhafte Eigenschaft, dass Modell und Implementierung zu jedem Zeitpunkt des Entwicklungsprozesses konsistent gehalten werden. Wird beispielsweise in einem Klassendiagramm eine neue Klasse definiert oder eine bestehende Klasse um ein Attribut oder eine Methode erweitert, so wird durch automatisches Generieren des entsprechenden Sourcecodes auch die Implementierung angepasst. Da auch die weiter oben erwähnten manuell in Klassendiagrammen eingefügten Abhängigkeiten durch einen entsprechenden Kommentar im Sourcecode verfügbar sind, reicht die alleinige Analyse des Sourcecodemodells aus, um dadurch alle für die Beurteilung der Testbarkeit relevanten Java-Abhängigkeiten erkennen und bestimmen zu können.

5.5.3 Auswertung des Sourcecodes über das *Together* Open API

Der Zugriff auf den Sourcecode erfolgt im *Together Open API* wie bereits beschrieben über die SCI-Schicht (siehe Abschnitt 5.4.1). Ausgangspunkt ist dort die Klasse `SciModelAccess`, das eine nach dem Interface `SciModel` implementierte Instanz der Sourcecode-Repräsentation eines *Together*-Projekts liefert:

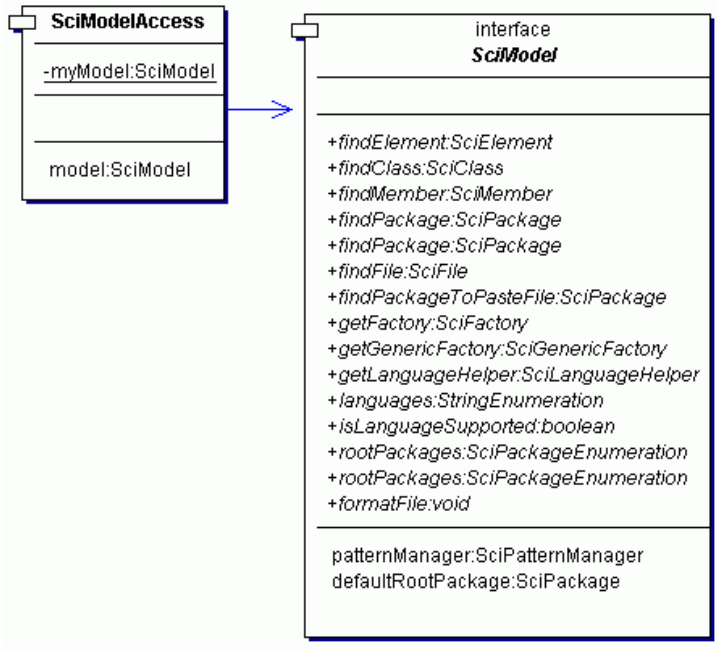


Abbildung 2: Zugriff auf die Sourcecode-Schicht des Together Open API

Die Methode `rootPackages()` im Interface `SciModel` liefert eines oder mehrere Pakete aus denen sich das Projekt zusammensetzt. Diese Pakete sind in *Together* als Verzeichnisse unter *Project Path* in den *Project Properties* aufgelistet, wobei es sich hier zumeist nur um ein einziges Paket handelt. Ausgehend von diesen Basispaketen, die wie alle Pakete das Interface `SciPackage` implementieren, können über die Methoden `subpackages()` und `classes()` rekursiv alle darin enthaltenen Pakete und Klassen auf der Suche nach Abhängigkeiten durchlaufen werden:

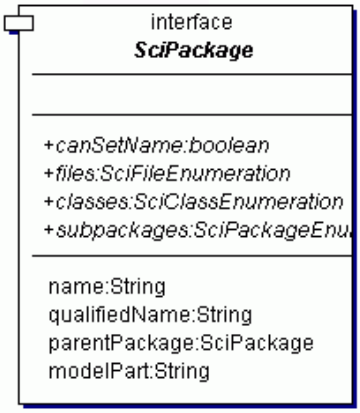


Abbildung 3: Interface SciPackage des Together Open API

Der komplette Sourcecode mit den darin enthaltenen Abhängigkeiten ist in `SciClass`-Objekte gegliedert. Ausgehend von den ermittelten Basispaketen erfolgt der Durchlauf durch sämtliche in den Paketen und ihren Unterpaketen als `SciClass`-Objekte verfügbaren Klassen. Auf Klassenebene startet die Untersuchung des Sourcecodes nach Abhängigkeiten. Der Programmcode dieser Klassen ist in weitere SCI-Elemente strukturiert. Eine Übersicht über die elementaren Bestandteile eines `SciClass`-Objekts gibt nachfolgende Skizze. Die Benennung der Elemente ist im übrigen identisch mit den Zugriffsmethoden des Interfaces `SciClass` auf diese Komponenten:

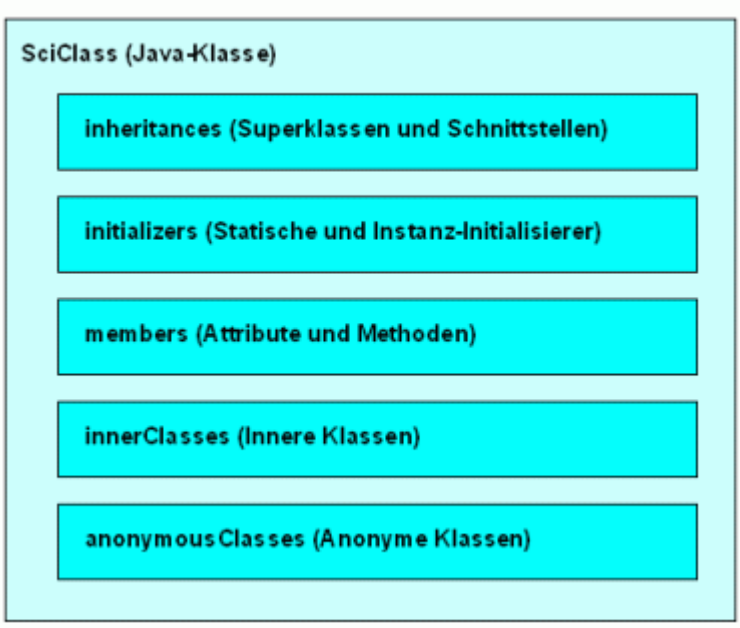


Abbildung 4: Elementare Bestandteile einer SciClass

Abhängigkeiten zu anderen Programmkomponenten können in allen oben aufgeführten Elementen vorhanden sein. Folglich müssen im Rahmen der Analyse sämtliche Elemente durchlaufen werden.

Für das Durchlaufen der Klassen und deren repräsentierenden Elementen stellt das *Together Open API* ein Interface zur Verfügung, das das Entwurfsmuster *Visitor* **[**Gamm94**]** realisiert. Dieses Interface findet Verwendung, um durch Zuweisung des Visitors an die Klassen Zugriff auf deren Inhalte einschließlich der oben beschriebenen Elemente der Klassen zu bekommen:

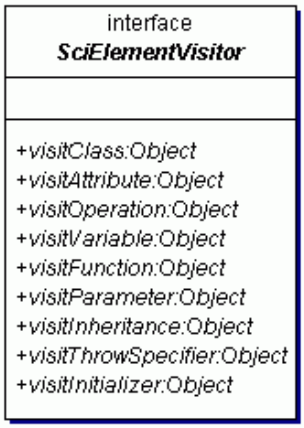


Abbildung 5: Interface SciElementVisitor

Teilweise ist es bereits auf dieser Ebene möglich (also bei der Untersuchung der Elemente mittels Implementierung der `visit...`-Methoden dieses Interfaces) Abhängigkeiten zu anderen Programmkomponenten zu identifizieren. In den meisten Fällen muss die Analyse allerdings noch einen Schritt weiter gehen. Dies soll nachfolgend am Beispiel des Durchlaufens von Methoden veranschaulicht werden.

Hierfür erfolgt zunächst ein Blick auf die Modellierung von Methoden im *Together Open API*. Jede Methode einer Klasseninstanz wird durch ein Objekt vom Typ `SciOperation` repräsentiert, das gleichzeitig noch weitere hierarchisch angeordnete Interfaces implementiert:

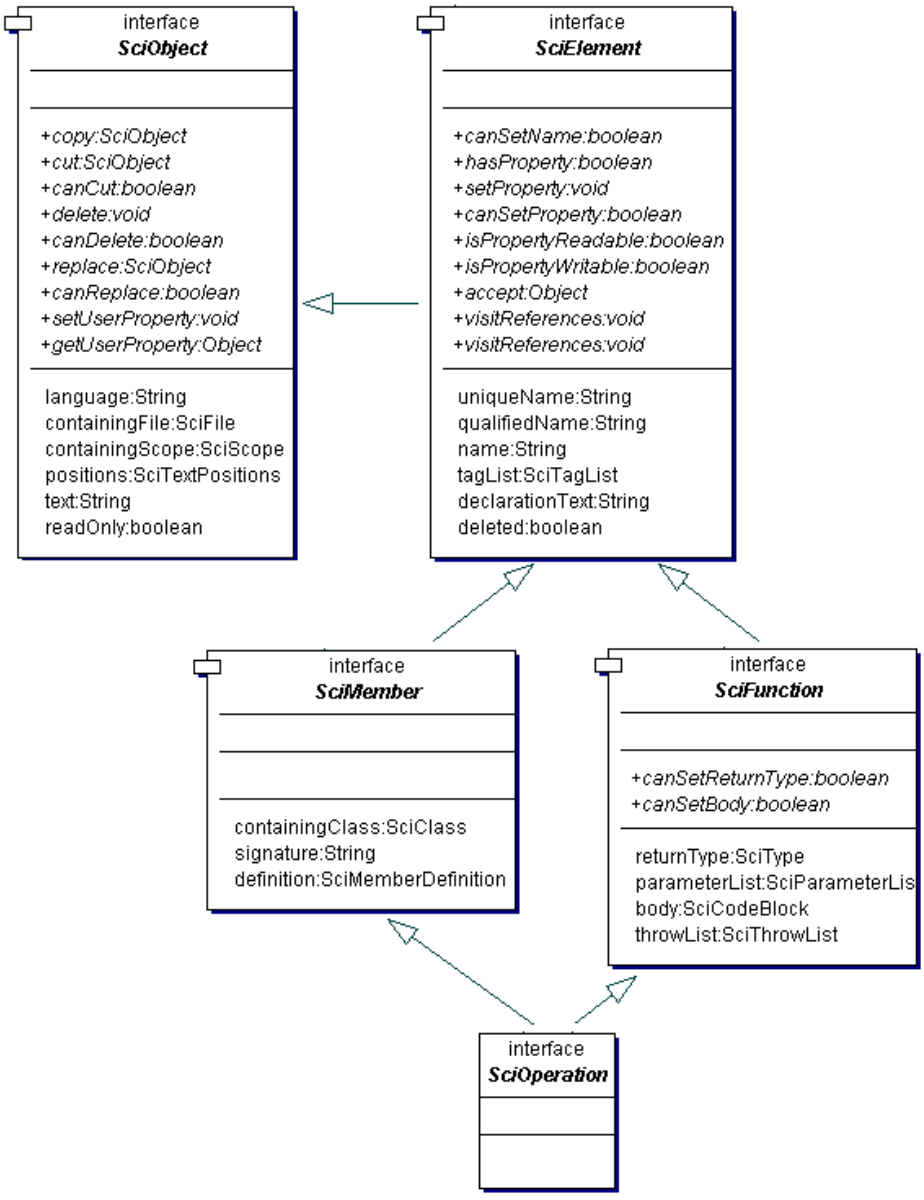


Abbildung 6: Repräsentation einer Methode im Together Open API

Mit Hilfe der von einer Methodeninstanz über die oben dargestellten Interfaces angebotenen Schnittstellen können im Rahmen der Implementierung der Methode

```
public Object visitOperation( SciOperation sciOperation)
```

des Interfaces `SciElementVisitor` beispielsweise Abhängigkeiten zu anderen Programmkomponenten aufgrund des Rückgabewerts der Methode, der übergebenen Parameter und der von der Methode geworfenen Ausnahmen bereits erkannt werden.

Um auch den Methodenrumpf nach Abhängigkeiten untersuchen zu können, stellt das Interface `SciFunction` diesen als Rückgabewert der Methode

```
public SciCodeBlock getBody()
```

in einem Objekt vom Typ `SciCodeBlock` zur Verfügung. Dieser Instanz können weitere vom *Together Open API* vorgesehene Visitoren zugewiesen werden, um die Suche nach Abhängigkeiten innerhalb des Methodenrumpfes fortsetzen zu können:

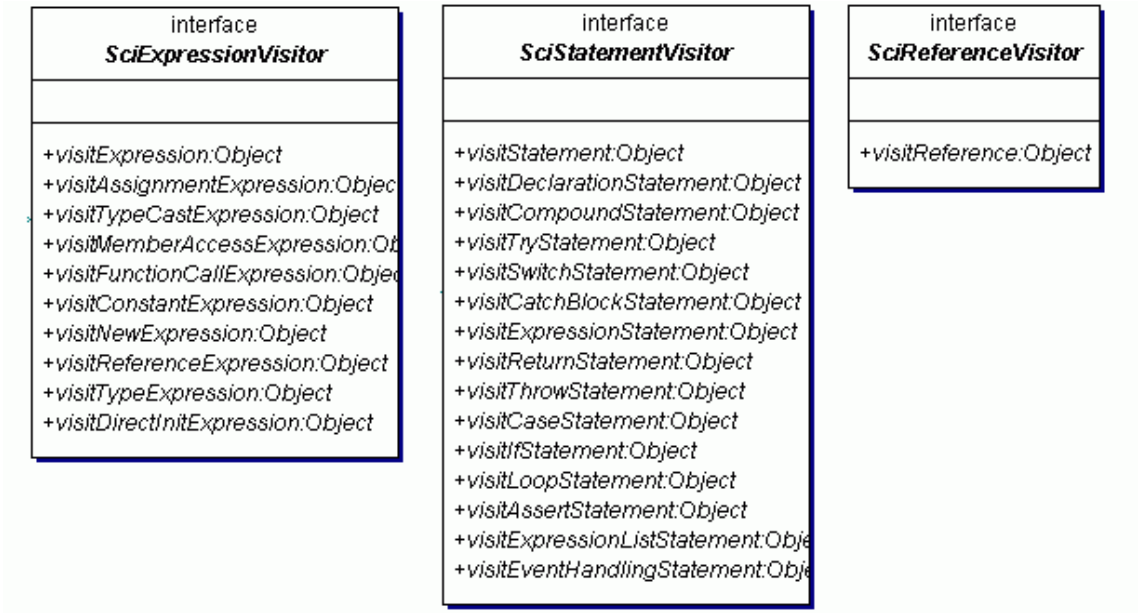


Abbildung 7: Weitere Visitor-Klassen im Together Open API

Die Identifizierung aller festgelegten Abhängigkeiten (siehe Abschnitt 5.5.1) kann über die Implementierung der in diesen Visitor-Interfaces zur Verfügung stehenden Methoden abgedeckt werden. Die beschriebene Vorgehensweise, die am Beispiel der Suche nach Abhängigkeiten in Methoden aufgezeigt wurde, kann analog für alle anderen Sourcecode-Bestandteile verwendet werden. Somit sind die Voraussetzungen geschaffen, darauf aufbauend den Entwurf der Teilkomponente zur Ermittlung der Abhängigkeiten erstellen zu können (siehe Abschnitt 6.3).

5.6 Erzeugung des Abhängigkeitsgraphen (T 3)

Für den Entwurf ist die Frage von Bedeutung, welche Schnittstellenbeschreibung *ImproveT* im Hinblick auf das Erzeugen und den Aufbau eines Abhängigkeitsgraphen vorsieht. Dies wird in den nächsten beiden Abschnitten beschrieben. Im Anschluss daran muss geklärt werden, in welcher Form jede einzelne in Abschnitt 5.5.1 bestimmte mögliche Abhängigkeit in Java-Programmen in geeigneter Weise in den Abhängigkeitsgraphen eingefügt werden kann. Dies wird die Basis für den späteren Entwurf (siehe Abschnitt 6.4) zur Umsetzung dieser Teilaufgabe sein.

5.6.1 Definition des Abhängigkeitsgraphen in *ImproveT*

Die Basis für alle Metrikberechnungen in *ImproveT* bildet ein Abhängigkeitsgraph. Zur Objekterzeugung eines Abhängigkeitsgraphen liefert im Paket `system` die Klasse `Factory` eine Instanz einer Implementierung des Interfaces `GraphFactory`, von der über eine gleichnamige *getter*-Methode eine Instanz einer Implementierung des Interfaces `ExtendedGraph` empfangen werden kann:

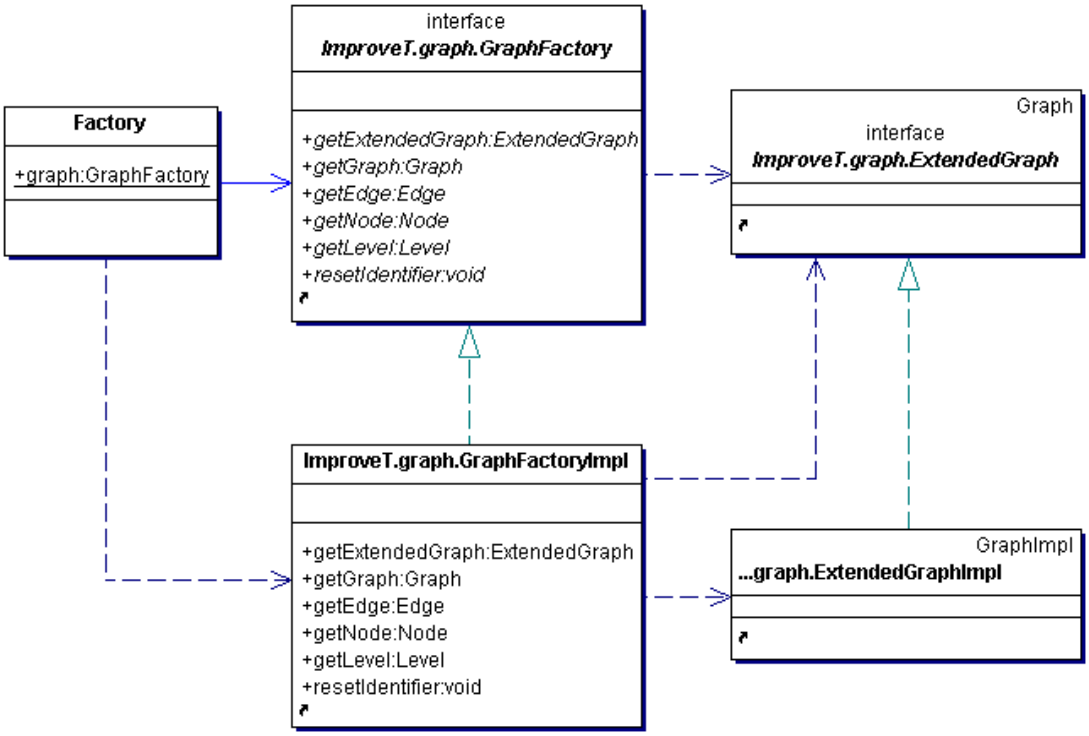


Abbildung 8: Klassendiagramm zur Erzeugung eines Abhängigkeitsgraphen

Die Modellierung der Abhängigkeiten eines objektorientierten Softwaresystems erfolgt im Abhängigkeitsgraphen von *ImproveT* auf verschiedenen Ebenen (z.B. Member- und Klassenebene). Auf jeder dieser Ebene werden die abhängigen sowie weitere für die Metrikberechnung relevante Programmkomponenten als Knoten des Graphen modelliert (z.B. Attribute und Methoden als Knoten auf der Memberebene). Eine Abhängigkeit zwischen zwei Komponenten wird durch eine die beiden Knoten verbindende Kante dargestellt.

Die verschiedenen Ebenen des Abhängigkeitsgraphen können ferner durch entsprechende Verweise untereinander verbunden werden. So können beispielsweise einem Knoten auf Klassenebene alle seine in der Memberebene als Knoten gespeicherten Attribute und Methoden in einer Datenstruktur als Unterknoten zugeordnet werden. Dasselbe gilt für die Zuordnung der Abhängigkeiten der tiefer liegenden Ebene zu der aggregierten Abhängigkeit auf der jeweils unmittelbar benachbarten höheren Ebene. Diese Vernetzung der einzelnen Graphebenen bietet die Möglichkeit, ausgehend von einer Abhängigkeit und den daran beteiligten Komponenten auf den höheren Ebenen eine verfeinernde Darstellung der verursachenden Abhängigkeiten bis hin zur untersten Ebene des Graphen zu erhalten.

5.6.2 Einfügen der Abhängigkeiten und Komponenten in den *ImproveT*-Graph

Nach der Definition der verschiedenen Ebenen des Abhängigkeitsgraphen durch mehrfachen Aufruf der Methode

```
void addLevel( int LevelNumber, String levelName);
```

des Abhängigkeitsgraphen, können auf den dadurch festgelegten Ebenen die dort anzusiedelnden Knoten (Komponenten) und Kanten (Abhängigkeiten) eingefügt werden.

Bei den Knoten und Kanten des Graphen handelt es sich um Instanzen, die die Interfaces `Node` und `Edge` als Typen implementieren. Nachfolgend Ausschnitte aus den Klassendiagrammen der beiden Typen. Die zwischen den Klassen und Interfaces vorhandenen Assoziationen und Abhängigkeiten spiegeln die Struktur des entstehenden Abhängigkeitsgraphen wieder:

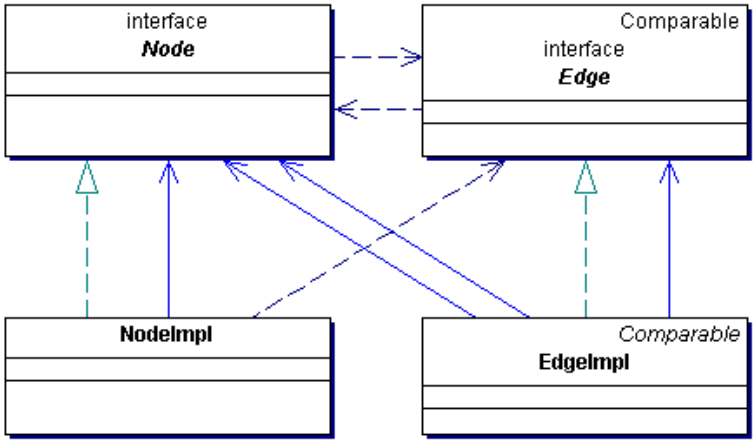


Abbildung 9: Klassendiagramme der Graphelemente Node und Edge

Für jede Abhängigkeit zwischen zwei Komponenten werden die beiden beteiligten Komponenten als Knoten und die Abhängigkeit als Kante in die entsprechende Ebene des Abhängigkeitsgraphen eingefügt. Für das Einfügen von Knoten und Kanten bietet eine Instanz eines Abhängigkeitsgraphen die beiden Methoden

```
Node getNode(int levelNumber, String qualifiedName, short type);
Edge getEdge(int levelNumber, String name, short type,
             Node tailNode, Node headNode);
```

an. Sofern der einzufügende Knoten oder die einzufügende Kante bereits existieren, liefern beide Methoden einen Verweis auf die bereits eingefügten Graphelemente. Ansonsten werden die Objekte in der Datenstruktur des Graphen neu angelegt und an den Aufrufer der beiden Methoden zurück geliefert.

Aus den Signaturen der beiden Methoden ist bereits zu erkennen, dass die Elemente des Graphen durch die Ebene, auf der sie sich befinden, sowie einem eindeutigen Namen und einem Typ identifiziert werden. Bei den Abhängigkeiten kommen ferner als Attribute noch die beiden an der Abhängigkeit beteiligten Komponenten hinzu. Beim Typ handelt es sich um die Art der Abhängigkeit (z.B. Aufruf einer statischen Methode, Erzeugung einer Instanz) sowie im Falle von Komponenten um

deren Charakterisierung (z.B. Klasse, Konstruktor, Attribut). Die zur Verfügung stehenden Typen sind im Interface `GraphTypes` von *ImproveT* definiert.

Um die im einleitenden Abschnitt bereits beschriebene Verbindung der verschiedenen Ebenen des Graphen modellieren zu können, stellen die Interfaces `Edge` und `Node` die Methoden

```
void addSubEdge(Edge subEdge);
void setSuperEdge(Edge superEdge);
void addSubNode(Node node);
void setSuperNode(Node node);
```

bereit.

Bei der Integration von *ImproveT* in das *Together ControlCenter* kommt ferner der Eigenschaft eine wichtige Bedeutung zu, beliebige Objekte (im Rahmen dieser Arbeit werden dies Objekte des *Together Open API* sein) als Inhalt der Kante bzw. des Knotens speichern zu können. Hierfür existiert in beiden Interfaces die Methode:

```
void addContent(Object content);
```

5.6.3 Modellierungsaspekte der Java-Abhängigkeiten im Graphen

Die bereits in Abschnitt 5.5.1 eingeführte Übersicht (Anhang B) über alle in einem Java-Programm möglichen Abhängigkeiten, wurde zur Lösung dieser Frage um die konkreten Modellierungsaspekte jeder Abhängigkeit erweitert.

Die wichtigsten Grundlagen der Modellierung sind nachstehend noch mal in komprimierter Form zusammengefasst:

- Abhängigkeiten zwischen zwei Klassenelementen (*members*) werden auf Memberebene als eigenständige Abhängigkeit gespeichert. Gleiche Typen von Abhängigkeiten in der Memberebene werden in der Klassenebene zu einer Abhängigkeit zusammengefasst.
- Eine Ausnahme hiervon bilden klassenlokale Abhängigkeiten: Abhängigkeiten zwischen Elementen derselben Klasse werden nur auf Memberebene im Graph gespeichert.
- Abhängigkeiten, die zwischen einem Klassenelement (*member*) und einer Klasse verlaufen, wie beispielsweise die Abhängigkeit einer Methode zum Klassentyp eines Parameters, werden nur auf Klassenebene im Graphen gespeichert. Auch hier existiert je Abhängigkeitstyp nur ein Eintrag. Um den Bezug zu den auslösenden Klassenelementen nicht zu verlieren, werden die Elemente als Unterknoten bei der Abhängigkeit gespeichert.
- Für jeden Konstruktor einer Klasse wird eine Abhängigkeit zu allen Instanzinitialisierern (*instance initializers*) der Klasse modelliert. Der Eintrag erfolgt nur auf Memberebene.
- Auf zusammengefasster Klassenebene (*summary class level*) existiert nur maximal eine Abhängigkeit zwischen abhängiger und aufgerufener Klasse. Alle Abhängigkeiten der Klassenebene werden dort zusammengefasst (ergänzt um die Abhängigkeiten innerer und anonymer Klassen)
- Sämtliche Klassen des Systems wie auch sämtliche Konstruktoren der Klassen werden als Knoten im Abhängigkeitsgraph gespeichert. Hierbei spielt es keine Rolle, ob eine Klasse oder ein Konstruktor an einer Abhängigkeit beteiligt ist oder nicht.

5.7 Export des Abhängigkeitsgraphen und Start der Metrikberechnung (T 4)

Nach der Erzeugung des Abhängigkeitsgraphen muss der Export des Graphen nach *ImproveT* erfolgen. Dort wird auf der Basis des Abhängigkeitsgraphen die Metrikberechnung gestartet. Die zu berechnenden Metriken sollen im Vorfeld vom Anwender ausgewählt werden können. Im nächsten Abschnitt werden die Schnittstellen von *ImproveT* zur Erledigung dieser Aufgaben beschrieben. Darauf aufbauend erfolgt der Entwurf der Teilkomponente zum Start der Metrikberechnung (siehe Abschnitt 6.5).

5.7.1 Analyseschnittstelle von *ImproveT*

Auf Basis des im Abschnitt 5.6 beschriebenen Abhängigkeitsgraphen erfolgt die Berechnung der in Abschnitt 1.3 vorgestellten Testbarkeits- und Reduktionsmetriken. Diese werden um zusätzliche Informationen ergänzt, die eine bessere Einordnung der Metriken im Hinblick auf die Testbarkeit des Systems ermöglichen.

Für die tatsächliche Berechnung von Abhängigkeits- und Komponentenmetriken sind die beiden Klassen `VersionAnalysis` und `NodeAnalysis` verantwortlich. Beide Klassen sind von der Java-Swing-Klasse `AbstractTableModel` abgeleitet. Als Ergebnis der Berechnung erhält man folglich ein Tabellenmodell, das zur Anzeige gebracht werden kann.

Der Programmablauf zum Start der Berechnung ist in beiden Klassen identisch:

- Erzeugung einer Instanz der Berechnungsklasse,
- Setzen des zu analysierenden Abhängigkeitsgraphen,
- Definition der zu berechnenden Metriken,
- Starten der Analyse.

Die Zuweisung des zu analysierenden Abhängigkeitsgraphen erfolgt jeweils mittels der Methode:

```
void setGraph(ExtendedGraph g);
```

Da die Berechnung der Metriken nur für eine bestimmte Ebene eines Abhängigkeitsgraphen erfolgt, muss im Graph die zu analysierende Ebene vor dem Starten der Berechnung explizit gesetzt werden. Jede zu berechnende Metrik wird durch die Instanz einer Klasse repräsentiert. Die verfügbaren Metrikklassen für Abhängigkeiten sind im Paket `metrics`, die verfügbaren Komponentenmetriken im Paket `nodemetrics` angesiedelt. Die Metrikinstanzen werden vor dem Start einer Metrikberechnung erzeugt und über die Methode

```
void addMetric(MetricSet metric);
```

den Analyseklassen hinzugefügt. Dort werden sie während der ganzen Laufzeit des Programmes in einer Datenstruktur vorgehalten.

Wurden die beiden Berechnungsklassen mit dem zu analysierenden Graphen und den zu berechnenden Metriken (siehe Abschnitt 0) konfiguriert, so kann die Analyse des Graphen und die Berechnung der Metriken gestartet werden. Zum Start der Berechnung dienen die beiden `analyze`-Methoden:

```
void analyzeVersions();
void analyzeNodes();
```

5.8 Anzeige der Testbarkeitsmetriken (T 5)

Nach Abschluss der Metrikberechnung sollen die *ImproveT*-Ergebnisse in *Together* abgerufen werden können. Dieser Abschnitt beschreibt die von den beiden zu integrierenden Systemen hierzu angebotenen Ausgabeschnittstellen. Dies wird Grundlage für den Entwurf der Ausgabekomponenten in Abschnitt 6.7 sein.

5.8.1 Ausgabeschnittstelle von *ImproveT*

Die ermittelten Abhängigkeits- und Komponentenmetriken sind, wie in Abschnitt 5.7.1 beschrieben, nach Abschluss der Analysen in den beiden Tabellenmodellen `VersionAnalysis` und `NodeAnalysis` gespeichert. *ImproveT* stellt zwei von der Java-Swing-Klasse `JTable` abgeleitete Tabellen zur Anzeige der Ergebnisse zur Verfügung: `VersionAnalysis` und `NodeAnalysisTable`

Beide Tabellen erwarten bei der Instantiierung die während der Analyse aufgebauten Tabellenmodelle (siehe Abschnitt 5.7.1) als Übergabeparameter. Diese ermöglichen den Zugriff auf die in den Tabellen angezeigten Kanten und Knoten des Abhängigkeitsgraphen. Hierzu stellen sie Methoden zur Verfügung, die für eine Zeile der Tabelle die dahinter liegende Kante oder den zugrunde liegenden Knoten liefern. Ferner kann zu jeder Spalte der Tabelle die Instanz der angezeigten Metrik vom Tabellenmodell abgerufen werden:

```
Edge getEdgeForRow(int row);
MetricSet getMetric(int col);
Node getNodeForRow(int row);
NodeMetric getMetric(int col);
```

Die beiden Graphelemente `Edge` und `Node` stellen weiterhin verschiedene Methoden zur Verfügung, um die in den vorhergehenden Abschnitten bereits beschriebene Struktur des Abhängigkeitsgraphen wieder auslesen zu können. So bieten beispielsweise die Verbindungen zwischen den verschiedenen Graphebenen die Möglichkeit zur Navigation innerhalb des Graphen, um eine detaillierte Anzeige der Abhängigkeiten zu realisieren. Um bei der Anzeige der Metrikergebnisse die unmittelbare Verbindung zu *Together*-Elementen vornehmen zu können, können ferner die als Inhalte von Kanten und Knoten des Graphen gespeicherten Verweise auf diese Elemente ausgelesen werden:

```
Collection contents();
Collection subEdges();
Collection subNodes();
Collection incomingEdges();
Collection outgoingEdges();
```

Da sowohl die in diesem Abschnitt beschriebenen Metriktabellen als auch ihre Tabellenmodelle in *Together* zur Anzeige der Metrikergebnisse verwendet werden können, werden diese bei der Anzeige der Testbarkeitsmetriken als Grundlage für den Entwurf der Ausgabekomponente dienen (siehe Abschnitt 6.7).

5.8.2 Graphische Anzeige der Abhängigkeiten (Ausgabeschnittstelle von *ImproveT*)

ImproveT bietet die Möglichkeit, den zu Beginn erzeugten Abhängigkeitsgraphen zu visualisieren. Die Visualisierung erfolgt über die Darstellung des Graphen als geometrischer Graph mit Knoten und Kanten in der Zeichenebene. Für die Darstellung des Graphen zeichnet die Klasse `TopologicalGraphView` verantwortlich. Für die vollständige Integration in eine externe Anwendung dient ferner die nach dem Listener-Muster vorgenommene Implementierung `GraphViewMouseHandler`:

Abbildung 10: ImproveT-Schnittstelle zur Anzeige des Abhängigkeitsgraphen

Nach der Erzeugung einer Instanz von `TopologicalGraphView` wird dieser Instanz ein Abhängigkeitsgraph zur Anzeige übergeben:

```
void setGraph(ExtendedGraph graph);
```

Über die Methoden `myShow()` und `startSort()` lässt sich die Darstellung des Graphen anstoßen. Die Anzeige des in der Darstellung aktiven Graphelements lässt sich ebenfalls über die öffentliche Schnittstelle der Klasse `TopologicalGraphView` steuern. Darüber hinaus stehen verschiedene Farbschemata für die Kolorierung des Graphen zur Verfügung. Jedes dieser Schemata hebt farblich bestimmte Testbarkeitsaspekte des Graphen hervor:

```
void setSelectedEdge(Edge e);
void setSelectedNode(Node n);
void setColorSchema(int schema)
```

Erwähnenswert ist ferner die Möglichkeit, Beobachter bei der Instanz registrieren zu können, die über Änderungen der Anzeige informiert werden. Da die aktuellen Anzeigeinformationen über die Schnittstelle ausgelesen werden können, ermöglicht dies bei der beabsichtigten Integration in das *Together ControlCenter* die verschiedenen Anzeigetableaus konsistent zu halten:

```
void addNodeSelectionObserver(MyObserver o);
void addEdgeSelectionObserver(MyObserver o);
Node getSelectedNode();
Edge getSelectedEdge();
```

5.8.3 Ausgabemöglichkeiten in Together

Als Ausgabeschnittstelle steht im *Together Open API* die IDE-Schicht zur Verfügung (siehe Abschnitt 5.4.1). Die beiden Klassen `IdeWindowManagerAccess` und `IdeMessageManagerAccess` gestatten Zugriff auf die Ausgabekomponenten, mit denen die Anzeige der Testbarkeitsmetriken und des Graphen erfolgen kann:

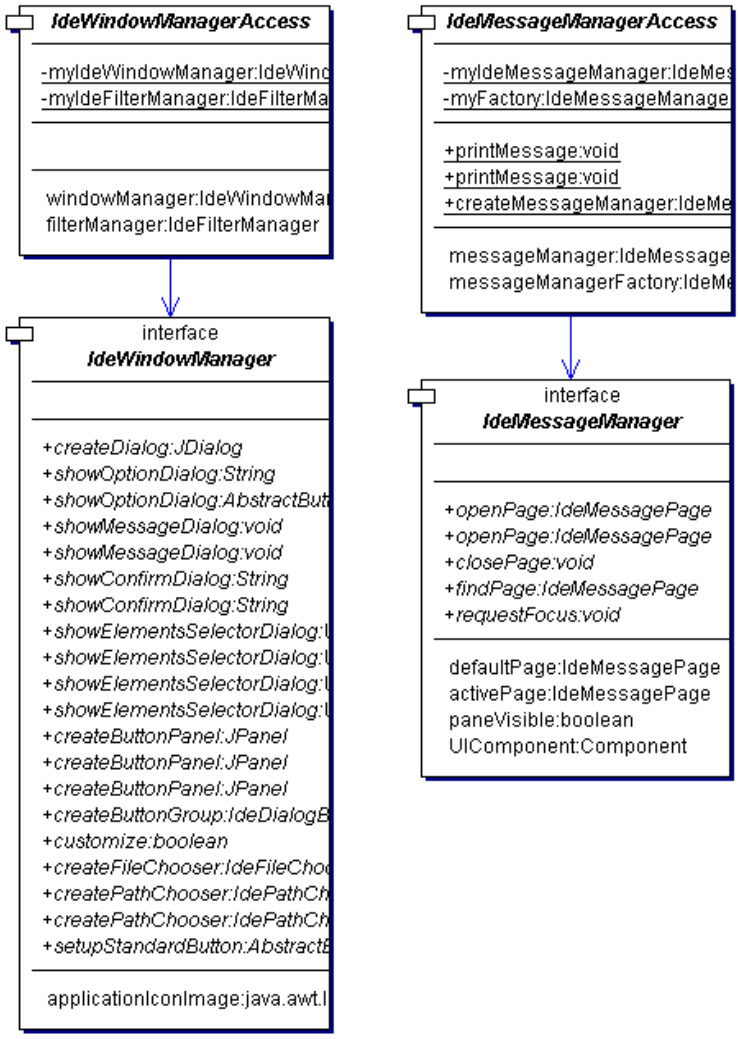


Abbildung 11: Wichtigste Ausgabekomponenten im Together Open API

Die Ausgabe der Metriktabellen erfolgt als Seite innerhalb des *Message Managers* von Together (`openPage`), die Anzeige des Abhängigkeitsgraphen kann als eigenständiges Fenster im *Window Manager* erfolgen (`createDialog`).

6 Entwurf

6.1 Einleitung

In der Analyse wurden die Grundlagen für den anstehenden Entwurf zur Erfüllung der Teilaufgaben gelegt. Aufbauend auf den dort gewonnenen Erkenntnissen werden in den folgenden Abschnitten nach und nach die einzelnen Teilkomponenten entworfen. Als Modellierungsplattform findet naheliegenderweise das *Together ControlCenter* Verwendung. Zur Notation der Entwürfe dient wiederum die *UML* [OUML00].

Zum besseren Verständnis wird nachstehend bereits vorab ein Überblick über die aus dem Entwurf zur Erledigung der Teilaufgaben resultierenden Komponenten gegeben. Neben der Zuordnung der Komponenten zu den Teilaufgaben ergibt sich ein erster Eindruck über die Struktur und die Abhängigkeitsbeziehungen der entworfenen Komponenten:

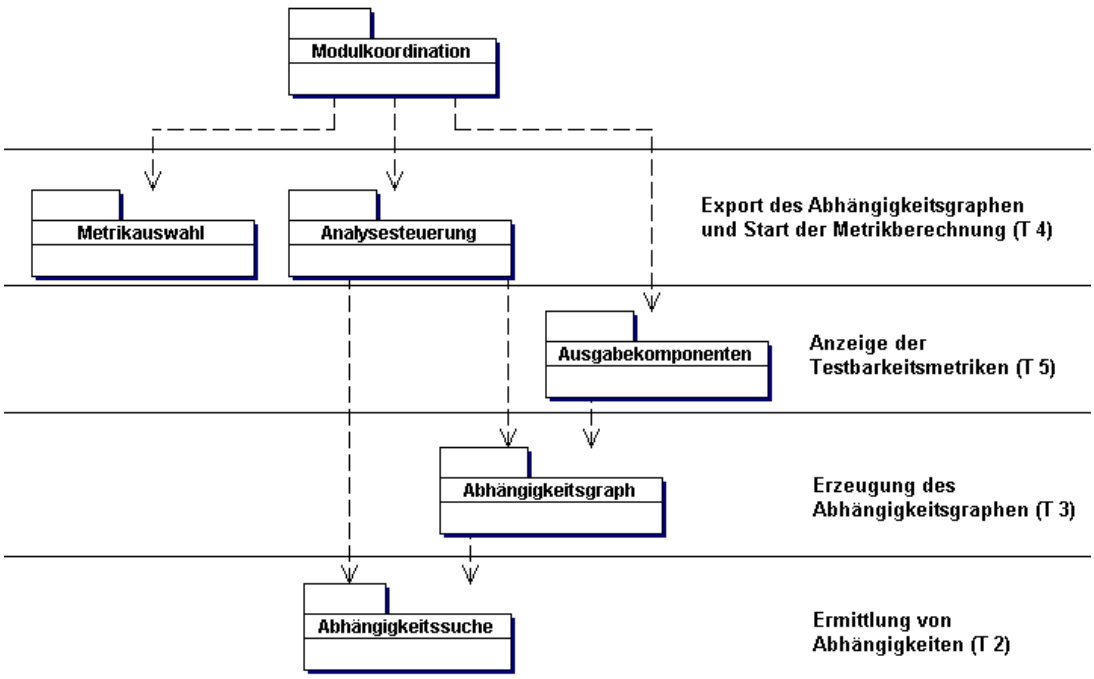


Abbildung 12: Überblick über die Teilkomponenten der Together-Erweiterung

Vor der Beschreibung der Teilkomponenten erfolgt zunächst noch die Klärung der Frage, wie die Integration der zu realisierenden Funktionalität in *Together* erfolgen kann.

6.2 Erweiterung von *Together* (class Design2Test)

Together bietet verschiedene Stufen der Integration einer Erweiterung über seine Programmierschnittstelle an. Ein *Together*-Modul kann bereits beim Start von *Together* geladen werden und dem Anwender damit bereits unmittelbar nach dem Hochfahren des System zur Nutzung zur Verfügung stehen. Alternativ bietet *Together* noch zwei verschiedene Formen der späteren Aktivierung des Moduls durch den Anwender nach dem Start des *ControlCenters*.

Durch die Aufgabenstellung war bereits vorgegeben, dass die Einbindung des Moduls in die Menüstruktur erfolgen sollte. Somit ergibt sich die Realisierung nach der oben beschriebenen ersten Variante. Hierzu schreibt das *Together Open API* die Implementierung der Hauptmodulklasse nach dem Interface `IdeStartup` vor:

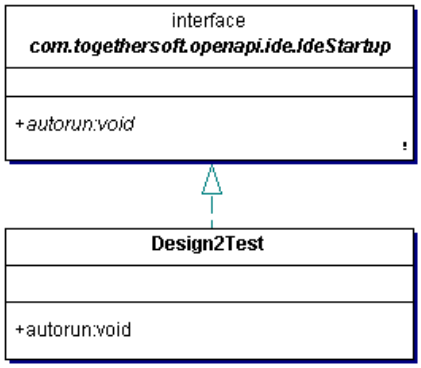


Abbildung 13: Integration als Together-Modul

Die Implementierung der `autorun()`-Methode wird die Erzeugung der Menüeinträge mit entsprechenden Ereignis-*Listenern* zum Start des Testbarkeitsmoduls vorsehen.

Aus dem abgebildeten Klassendiagramm lässt sich im Übrigen eine weitere Entwurfsentscheidung ablesen. Der Name der realisierten Together-Erweiterung wird *Design2Test* lauten.

6.3 Komponente Abhängigkeitssuche (package search)

Beim Entwurf der Suchkomponente liegt der Fokus auf der Wiederverwendbarkeit des Moduls. Zur Ablaufsteuerung und für den externen Zugriff auf das Analyse-Paket dient die Klasse `DependencySearcher`. Die Konfigurierbarkeit und damit Wiederverwendbarkeit dieser Klasse auch in anderen Anwendungsszenarien gewährleisten die Interfaces `DependencySearchListener` und `SearchScope`:

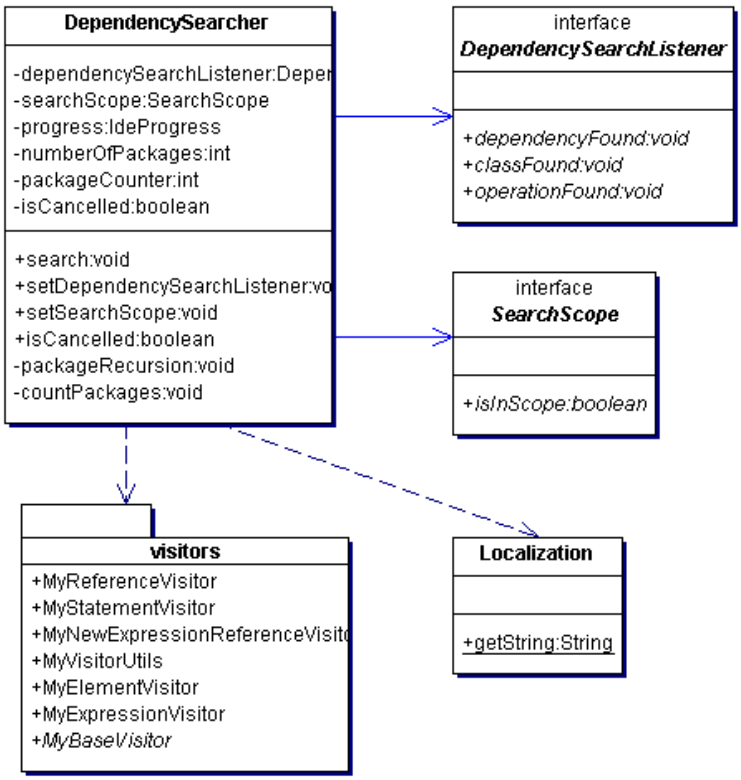


Abbildung 14: Klassendiagramm des Moduls zur Suche nach Abhängigkeiten (package search)

Eine Implementierung des Interfaces `DependencySearchListener` ist dafür verantwortlich, auf der Grundlage der ermittelten Abhängigkeiten den Abhängigkeitsgraphen für *ImproveT* zu erzeugen. Die für den Anwender vorzusehende Möglichkeit, im Startdialog bestimmte Klassen und Pakete von der Sourcecode-Analyse des Projekts auszuschließen, ist Inhalt der Implementierung des `SearchScope`-Interfaces.

Die eigentliche Suche und Ermittlung der Abhängigkeiten erfolgt, wie in der Analyse bereits ausgearbeitet (siehe Abschnitt 5.5.3), innerhalb der Klassen des `visitors`-Pakets. Während der `DependencySearcher` noch für das Durchlaufen des Sourcecodes des kompletten Projekts verantwortlich ist, übergibt er die Analyse für jede im Projekt gefundene Klasse an eine Instanz der Klasse `MyElementVisitor`. Diese delegiert die Zuständigkeit für die Analyse der einzelnen Klassenbestandteile gegebenenfalls an die anderen *visitor*-Klassen des Pakets:

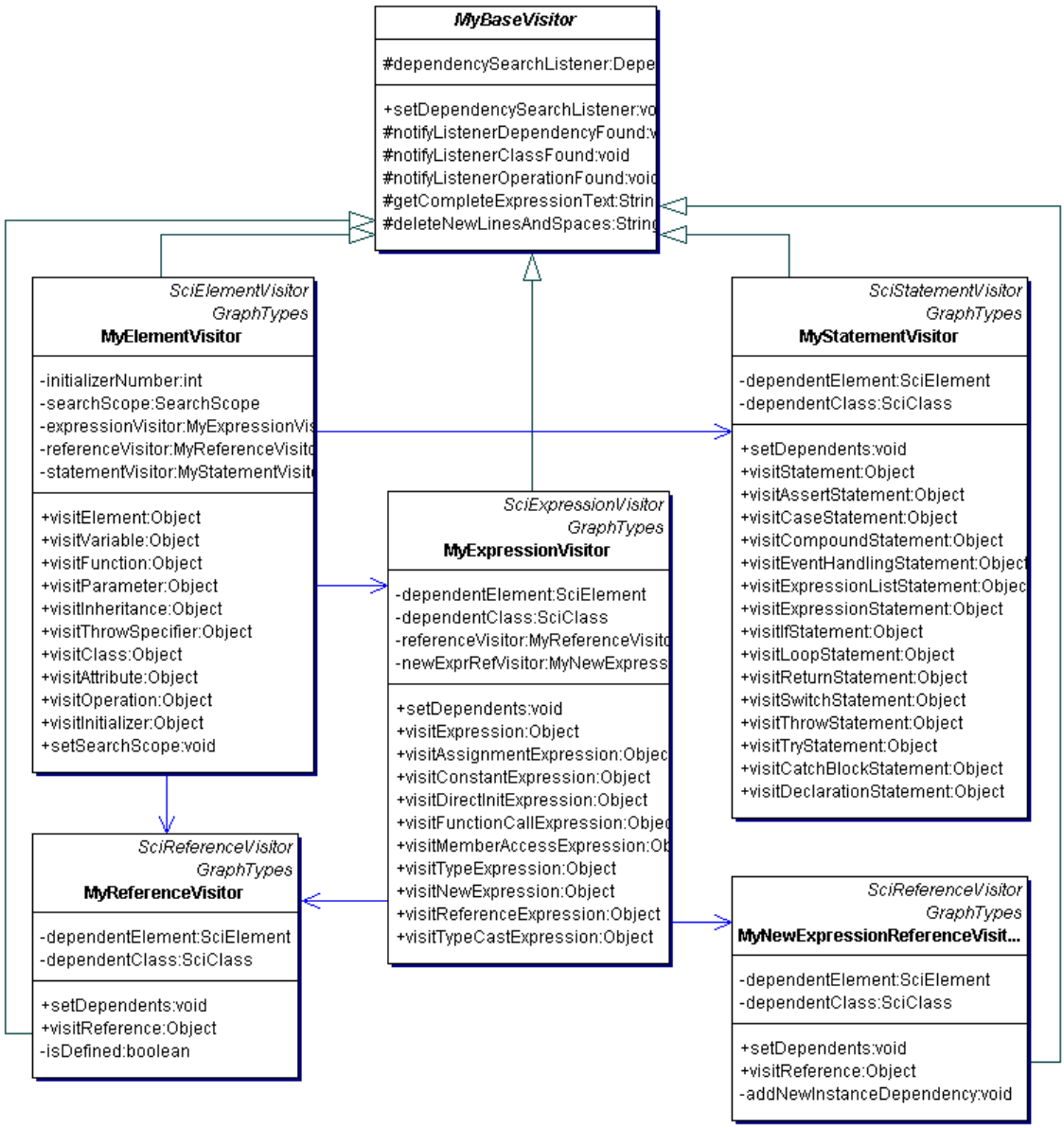


Abbildung 15: Übersicht über die Struktur der implementierten visitor-Klassen

Der Ablauf beim Start einer Suche nach Abhängigkeiten in einem *Together*-Projekt stellt sich zusammengefasst wie folgt dar:

- Erzeugung einer Instanz eines `DependencySearcher`.
- Erzeugung einer das Interface `DependencySearchListener` implementierenden Instanz und Registrierung dieser Instanz beim `DependencySearcher`.
- Erzeugung einer das Interface `SearchScope` implementierenden Instanz und Registrierung beim `DependencySearcher`.
- Starten der Abhängigkeitssuche durch Aufruf der Methode `search()`.

6.4 Komponente Abhängigkeitsgraph (package data)

Zentrale Klasse dieser Teilkomponente ist die als Einzelstück modellierte Klasse `DependencyGraph`. Sie hält als Attribut eine Instanz des zu exportierenden Abhängigkeitsgraphen vom importierten *ImproveT*-Typ `ExtendedGraph`. Ferner implementiert sie das Interface `DependencySearchListener` der oben entworfenen Suchkomponente (siehe Abschnitt 6.3). Durch die Anmeldung als *Listener* beim `DependencySearcher` wird die Klasse über alle existierenden Abhängigkeiten, Klassen und Operationen des zu analysierenden Softwareprojekts durch Aufruf der zu implementierenden *Listener*-Methoden `dependencyFound()`, `classFound()` und `operationFound()` informiert:

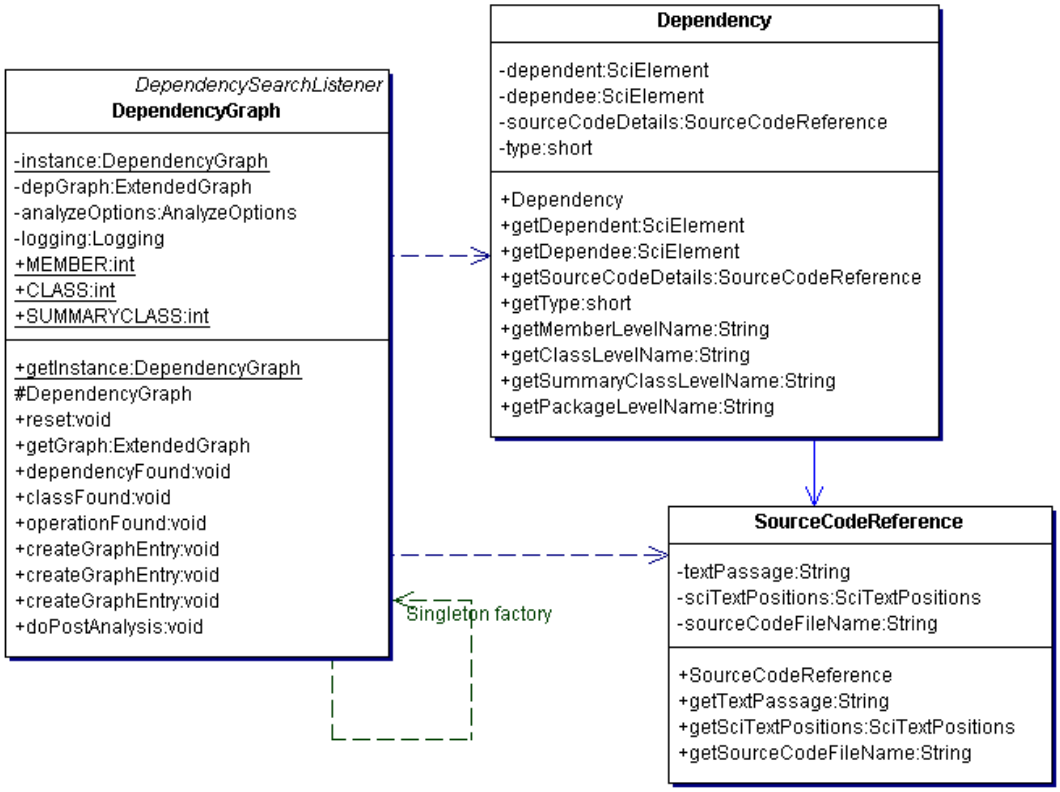


Abbildung 16: Klassendiagramme des DependencyGraph-Pakets (package data)

Anhand dieser Informationen zeichnet das Einzelstück der Klasse `DependencyGraph` in den Methoden `createGraphEntry()` verantwortlich für die korrekte Erstellung der Einträge in den nach *ImproveT* zu exportierenden Abhängigkeitsgraphen. Sie besitzt ein Attribut vom Interface-Typ `AnalyzeOptions` über das die Relevanz der Zielkomponenten von Abhängigkeiten geprüft werden kann. Eine Instanz von diesem Typ wird bei Aufruf der Methode `reset()` als Parameter übergeben. Der Aufruf dieser Methode muss jeder neuen Testbarkeitsanalyse vorausgehen. In ihr wird unter anderem für jeden Durchlauf eine neue `ExtendedGraph`-Instanz erzeugt und initialisiert.

Für jede ermittelte und tatsächlich relevante Abhängigkeit wird eine Instanz vom Typ `Dependency` erzeugt. Der Bezug zur Quelle der Abhängigkeit im Sourcecode wird in einer Instanz der Klasse `SourceCodeReference` als Attribut der Abhängigkeit verwaltet. Diese Referenz zum Sourcecode wird bei der Anzeige der Abhängigkeitsmetriken zur Navigation zum Ursprung der Abhängigkeit benötigt.

6.5 Komponente Analysesteuerung (package analysis)

Als zentrales Element dieser Teilkomponente wird die Klasse `ComputeMetrics` vorgesehen, die neben dem eigentlichen Anstoßen der Metrikberechnung in *ImproveT* auch die Erzeugung des Abhängigkeitsgraphen und die der Grapherstellung obligatorisch vorausgehende Suche nach Abhängigkeiten steuert. Den Zugriff auf die in Abschnitt 5.7.1 beschriebenen *ImproveT*-Module `VersionAnalysis` und `NodeAnalysis` verkapseln die beiden Klassen `VersionAnalyzer` und `NodeAnalyzer`:

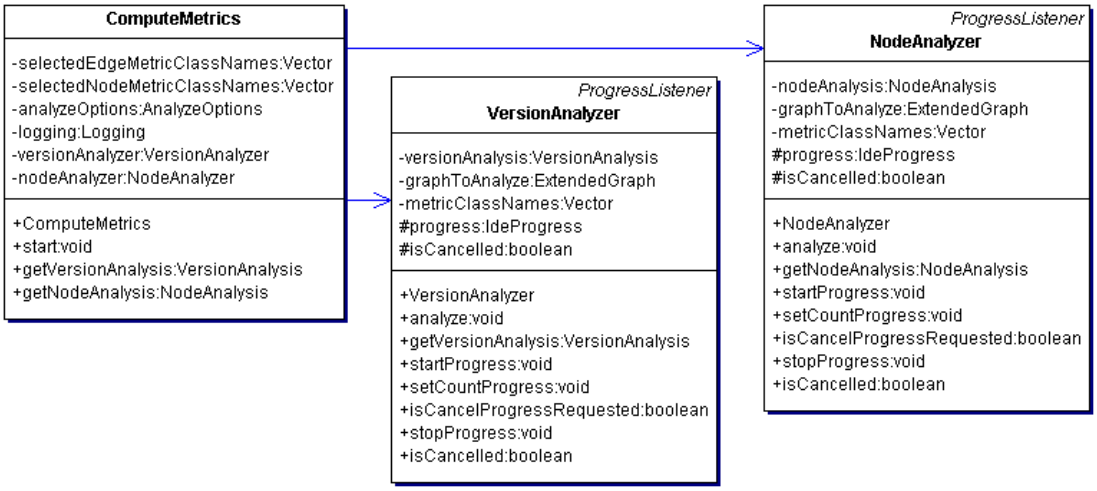


Abbildung 17: Benutzerschnittstelle zur Auswahl der Testbarkeitsmetriken

Nachdem der zu analysierende Abhängigkeitsgraph vorliegt, erzeugt `ComputeMetrics` die beiden Instanzen der `Analyzer`-Klassen. Diese sorgen für den Export des Graphen und den Start der Metrikberechnung in *ImproveT*, indem sie die in Abschnitt 5.7.1 beschriebenen Aufgaben übernehmen:

- Erzeugung von Instanzen der *ImproveT*-Klassen `VersionAnalysis` bzw. `NodeAnalysis`,
- Zuweisung des ermittelten Abhängigkeitsgraphen,
- Erzeugung und Zuweisung der zu berechnenden Metriken,
- Starten der Testbarkeitsanalysen.

6.6 Komponente Metrikauswahl (package selection)

Die Anforderungsspezifikation sieht die Möglichkeit vor, die im Rahmen der Testbarkeitsanalyse zu ermittelnden Metriken vor dem Start der Analyse durch den Anwender bestimmen zu lassen. Um nicht mit der Benutzeroberfläche (also dem *Look and Feel*) des *Together ControlCenters* zu brechen, erfolgt bei der Implementierung des Auswahldialogs die Orientierung an den in *Together* bereits integrierten Audit- und Metrikmodulen. Die auszuwählenden Metriken werden folglich in Tabellenform angeboten. Ferner stehen modale Unterdialoge zum Speichern und Laden von Metriken sowie zur Einstellung von Optionen bei der Grapherstellung (siehe Abschnitt 6.4) zur Verfügung:

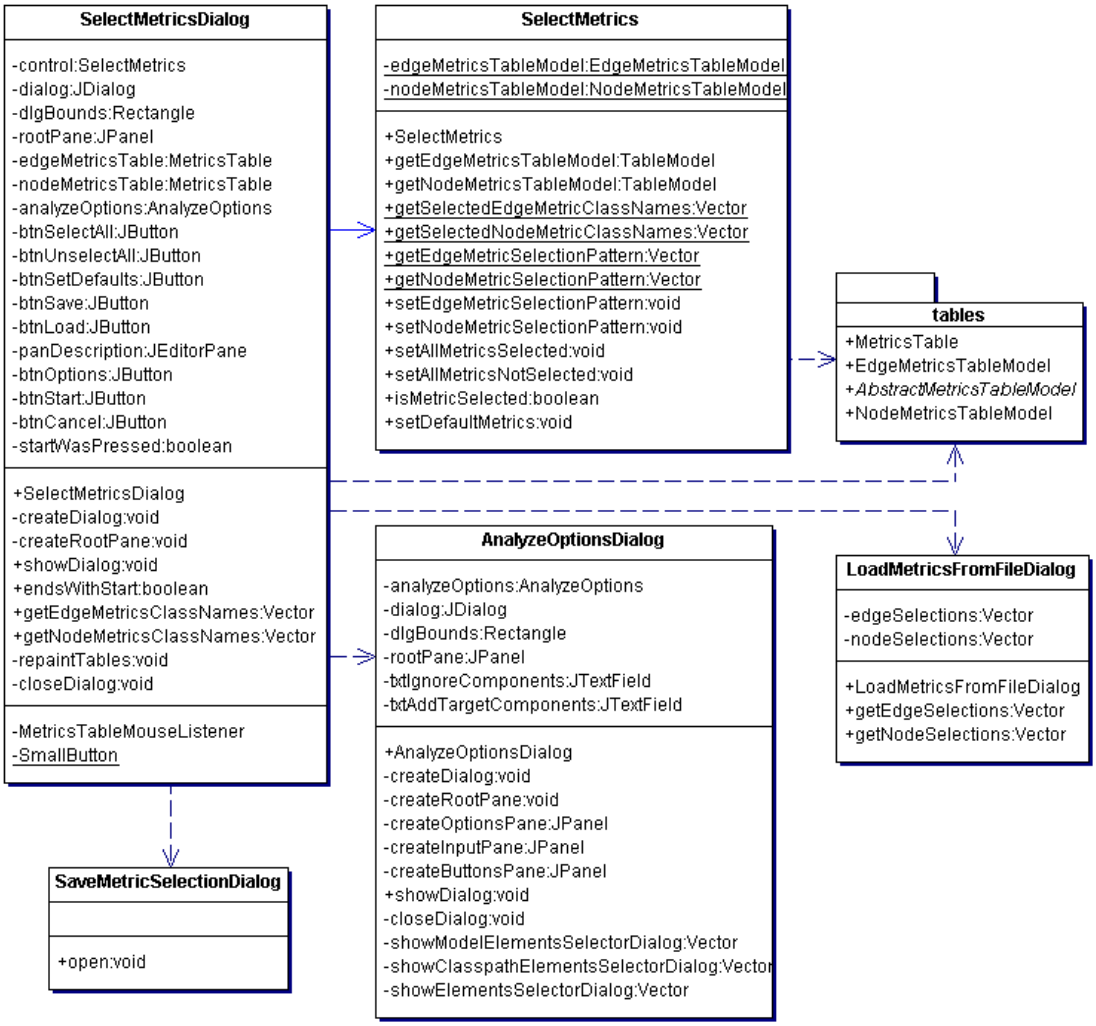


Abbildung 18: Benutzerschnittstelle zur Auswahl der Testbarkeitsmetriken

Bei der Klasse `SelectMetrics` handelt es sich um die zur Schnittstellenklasse `SelectMetricsDialog` gehörende Kontrollklasse. Sie stellt Zugriffsmethoden zur Verfügung, um die zuletzt ausgewählten Metriken abfragen zu können. Im Unterpaket `tables` erfolgt die Bereitstellung der zur Auswahl vorgesehen Metriken.

Auf die Beschreibung des Designs der Oberflächen wird innerhalb dieser Arbeit verzichtet. Das Aussehen der entworfenen Benutzerschnittstellen kann dem Benutzerhandbuch entnommen werden (siehe Anhang A).

6.7 Ausgabekomponenten (package results)

6.7.1 Einleitung

Nach Abschluss der Metrikberechnung können aus *ImproveT* die Ergebnisse dieser Berechnung abgerufen werden. Dieser Abschnitt beschreibt den Entwurf zur Präsentation der Ergebnisse in den Ausgabekomponenten des *Together ControlCenters.* Die für den Entwurf benötigten Schnittstellen wurden in Abschnitt 5.8 beschrieben.

6.7.2 Entwurf der Ausgabeschnittstelle des Moduls (Überblick)

Einen einleitenden Überblick über den für diesen Programmteil erstellten Entwurf gibt das Klassendiagramm des Paketes `results`, in dem sämtliche vorgesehenen Ausgabekomponenten zusammengefasst sind:

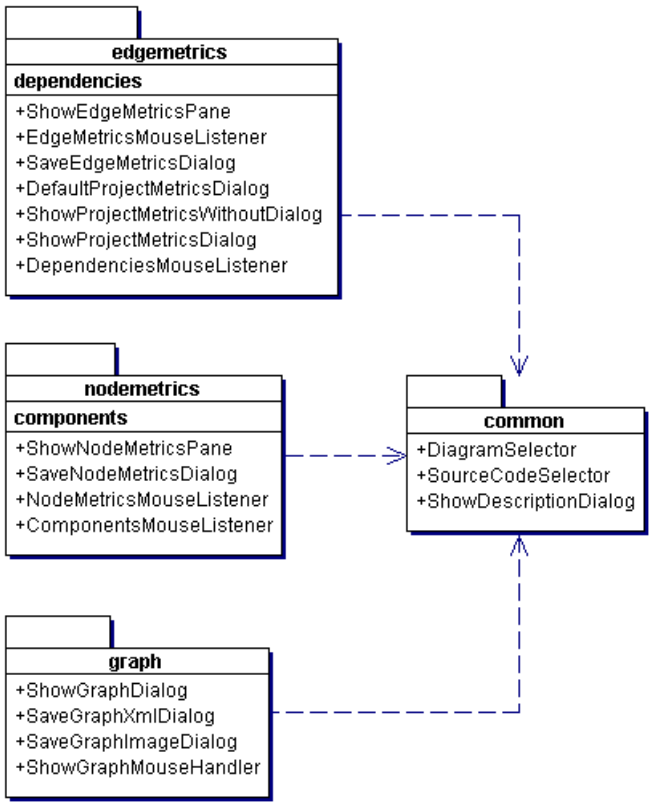


Abbildung 19: Das Ausgabepaket aller Ergebnisse (package results)

Die Struktur des Pakets gibt bereits Aufschluss über die vorgesehenen Ausgabekomponenten. Die drei am linken Rand des Diagramms angeordneten Unterpakete skizzieren jeweils eine der durch die Aufgabenstellung vorgegebenen Benutzerschnittstellen:

- **`edgemetrics`**
  Anzeige der Abhängigkeiten, der für die jeweiligen Abhängigkeiten ermittelten Metriken und der Codefragmente, durch die die Abhängigkeiten verursacht werden. Ferner Anzeigemodule für die aktuellen Projektmetriken und die durch Entfernung von Abhängigkeiten theoretisch erreichbaren Projektmetriken.
- **`nodemetrics`**
  Anzeige der Komponenten des analysierten Softwareprojekts und der für diese Komponenten ermittelten Metriken sowie eine Auflistung aller mit einer Komponente über Abhängigkeiten verbundenen Komponenten.
- **`graph`**
  Graphische Darstellung des Abhängigkeitsgraphen in verschiedenen Schemata zur Veranschaulichung der ermittelten Ergebnisse.

Das Paket `common` stellt darüber hinaus Hilfsklassen zur Navigation in die *Together* Anzeigekomponenten und zur Anzeige von Hilfetexten zu den Metriken zur Verfügung.

Bezüglich der Oberflächengestaltung erfolgt wiederum die Orientierung am *look and feel* von *Together* und seinen *Quality Assurance*-Modulen (*Audits* und *Metrics*). Die Ausgabe deren Ergebnisse erfolgt in der *Message Pane* von *Together*, so dass auch die Darstellung der ermittelten Testbarkeitsmetriken in der *Message Pane* erfolgt. Die in der Anzeige zur Verfügung stehenden Funktionalitäten werden analog zu der *Together*-Realisierung über Kontextmenüs angeboten. Für die graphische Anzeige wird ein separates Dialogfenster entworfen, das ebenfalls über Kontextmenüs bedient werden kann. Die aufgrund des Entwurfs tatsächlich realisierten Benutzeroberflächen können dem Benutzerhandbuch entnommen werden (Anhang A).

6.7.3 Teilentwurf Anzeige der Abhängigkeitsmetriken

Das nachstehende Klassendiagramm liefert einen Überblick über die Realisierung der Anzeige der Abhängigkeitsmetriken:

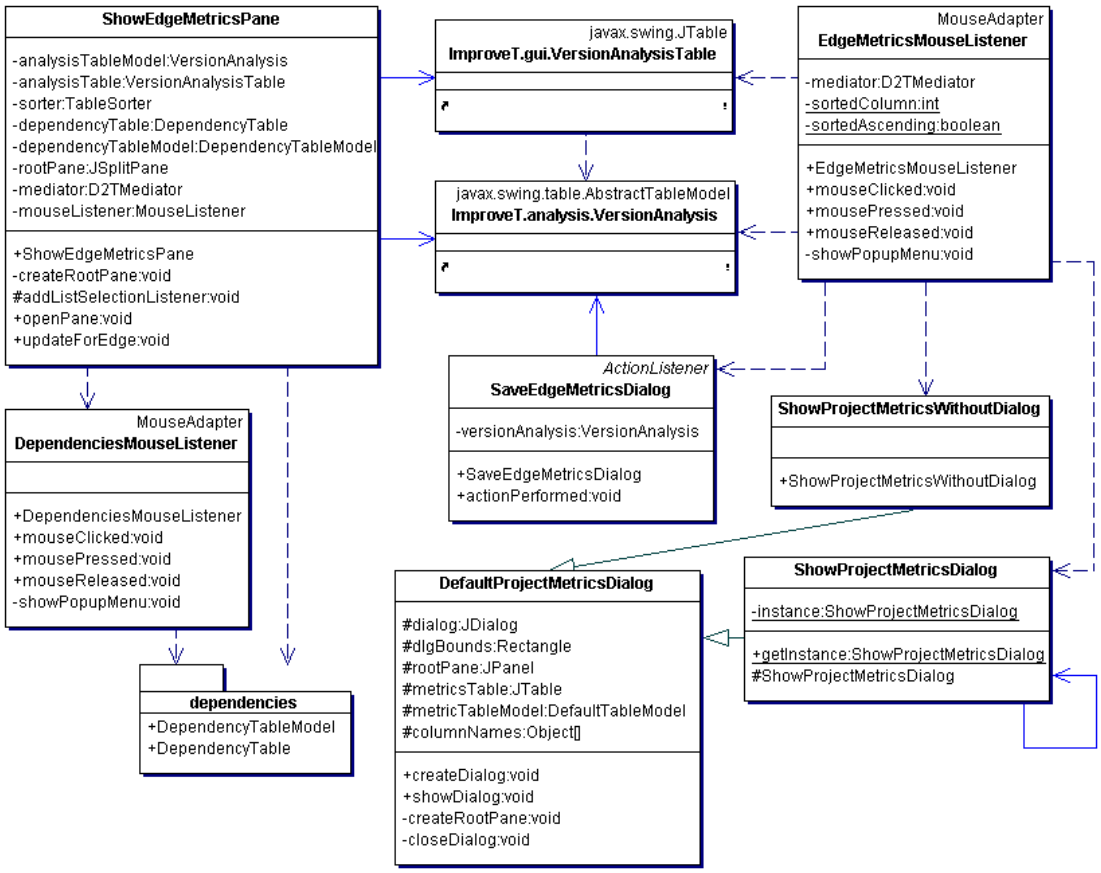


Abbildung 20: Anzeige der Komponentenmetriken (package nodemetrics)

Die Anzeige der Abhängigkeitsmetriken wird gesteuert durch die Klasse `ShowEdgeMetricsPane`. Sie bildet den Rahmen für die von *ImproveT* gelieferten Ergebnisse (`VersionAnalysisTable` und `VersionAnalysis`) und die Anzeige der Programmteile, die Auslöser von Abhängigkeiten sind (Paket `dependencies`). Die Klasse `EdgeMetricsMouseListener` ist verantwortlich für das Bereitstellen des Kontextmenüs in der Metrikanzeige, die Klasse `DependenciesMouseListener` realisiert das Kontextmenü in der Anzeige der Auslöser von Abhängigkeiten. Die übrigen Klassen des Pakets decken Teile der in den Kontextmenüs angebotenen Funktionalität ab.

6.7.4 Teilentwurf Anzeige der Komponentenmetriken

Das nachstehende Klassendiagramm liefert einen Überblick über die Realisierung der Anzeige der Komponentenmetriken:

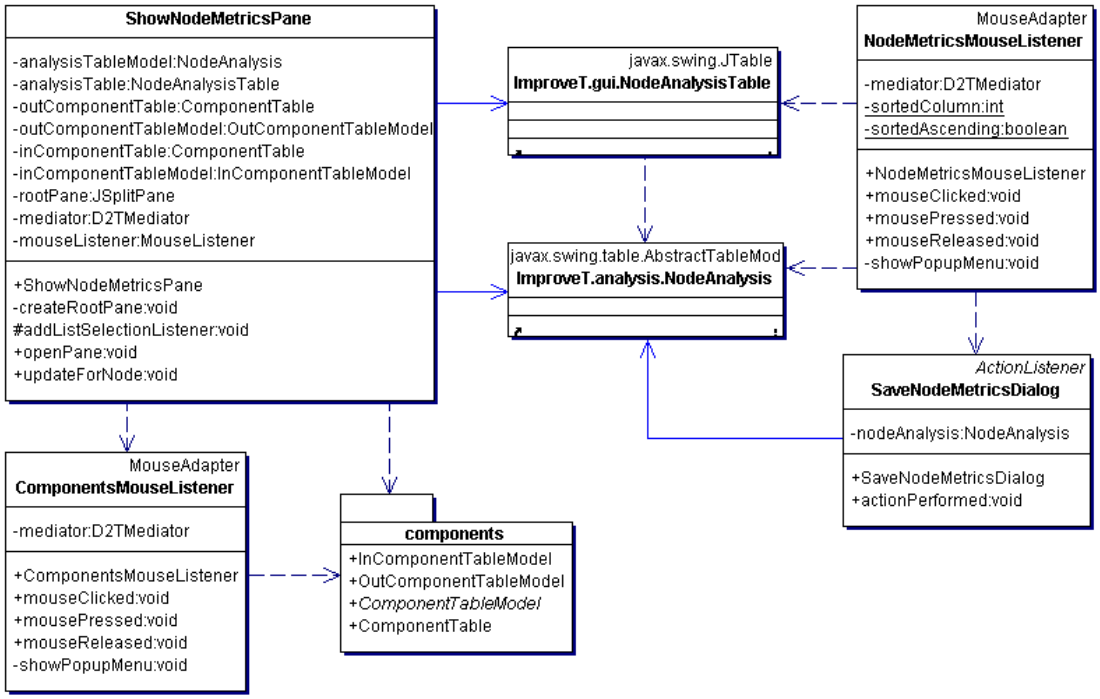


Abbildung 21: Anzeige der Komponentenmetriken (packageresults. nodemetrics)

Für die Anzeige der Komponentenmetriken zeichnet die Klasse `ShowNodeMetricsPane` verantwortlich. Sie bildet den Rahmen für die von *ImproveT* gelieferten Ergebnisse (`NodeAnalysisTable` und `NodeAnalysis`) und die Anzeige der Komponenten, die als Ziel oder Ursprung von Abhängigkeiten mit der aktuell ausgewählten Komponente verbunden sind (Paket `components`). Die Klasse `NodeMetricsMouseListener` ist verantwortlich für das Bereitstellen des Kontextmenüs in der Metrikanzeige, die Klasse `ComponentsMouseListener` realisiert das Kontextmenü in der Anzeige der verbundenen Komponenten. In der Klasse `SaveNodeMetricsDialog` ist das Speichern der ermittelten Metrikwerte realisiert.

6.7.5 Teilentwurf graphische Anzeige der Abhängigkeiten

Das nachstehende Klassendiagramm liefert einen Überblick über die Realisierung der graphischen Darstellung der Abhängigkeiten:

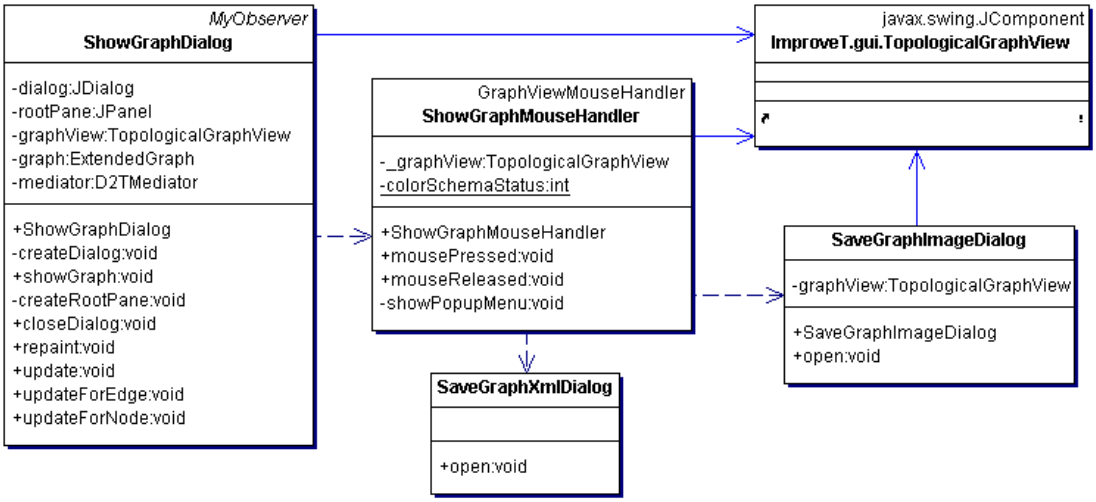


Abbildung 22: Graphische Darstellung der Abhängigkeiten (package results.graph)

Die Klasse `ShowGraphDialog` bindet die von *ImproveT* bereit gestellte Graphanzeige (`TopologicalGraphView`) in *Together* ein. Die Klasse `ShowGraphMouseHandler` zeichnet verantwortlich für das Bereitstellen des Kontextmenüs in der Graphanzeige, die übrigen Klassen realisieren die zur Speicherung des Graphen im Kontextmenü angebotene Funktionalität.

6.8 Komponente Modulkoordination (class D2TMediator)

Aus den verschiedenen Details der die funktionalen Anforderungen beschreibenden Anwendungsfälle (siehe Abschnitt 4.2) ergibt sich ein komplexes Zusammenwirken der einzelnen Teilmodule:

- Die Navigation aus der Anzeige der Abhängigkeitsmetriken in die Anzeige der Komponentenmetriken und umgekehrt soll möglich sein. Wählt der Anwender ausschließlich aus einem der beiden Bereiche Metriken zur Berechnung aus, so soll auch nur eines der beiden Anzeigefenster existieren.
- Die graphische Darstellung der Abhängigkeiten ist optional aus den Metrikanzeigen aufrufbar. Sobald sie zum Aufruf kommt, sollen die in den Metriktabellen angezeigten Abhängigkeiten und Komponenten mit den in der Graphanzeige selektierten Graphelementen übereinstimmen.
- Aus allen Anzeigekomponenten soll auf der Basis der aktuell selektierten Elemente die Navigation in den Texteditor oder den Diagrammmanager von *Together* möglich sein.
- Aus der Anzeige der Testbarkeitsmetriken soll der erneute Start der Metrikanalyse möglich sein. Dies soll entweder auf Basis der zuletzt vom Anwender getroffenen Auswahlen oder durch eine erneute Abfrage der zu berechnenden Metriken und Analyseoptionen möglich sein.
- Die Anzeige muss um weitere Komponenten für die Anzeige der Metriken auf Paketebene erweiterbar sein. Ein erster Entwurf hierzu wurde aufgrund offener Fragen bei der Berechnung der Metriken wieder verworfen.

Um dieses Zusammenspiel der genannten Teilkomponenten sicher zu stellen, mussten mehrere Entwurfsvarianten (siehe Beispiel 6.8.1) wieder verworfen werden, nachdem zu erkennen war, dass sie zu einem unübersichtlichen, fehleranfälligen und wartungsunfreundlichen Design der *Together*-Erweiterung geführt hätten.

Der zum Erfolg führende abschließende Entwurf sieht die Verwendung des Entwurfsmusters *Mediator* **[**Gamm94**]** für die Koordination der einzelnen Teilmodule der Erweiterung vor. Um den vom Mediator koordinierten Instanzen den Rückgriff auf die Mediatorinstanz zu ermöglichen, kommt das *Observer-Pattern* **[**Gamm94**]** zur Anwendung. Hierzu übergibt sich der Mediator bei der Erzeugung einer zu verwaltenden Programmkomponente selbst als Beobachter. Das resultierende Klassendiagramm zeigt nachstehende Abbildung:

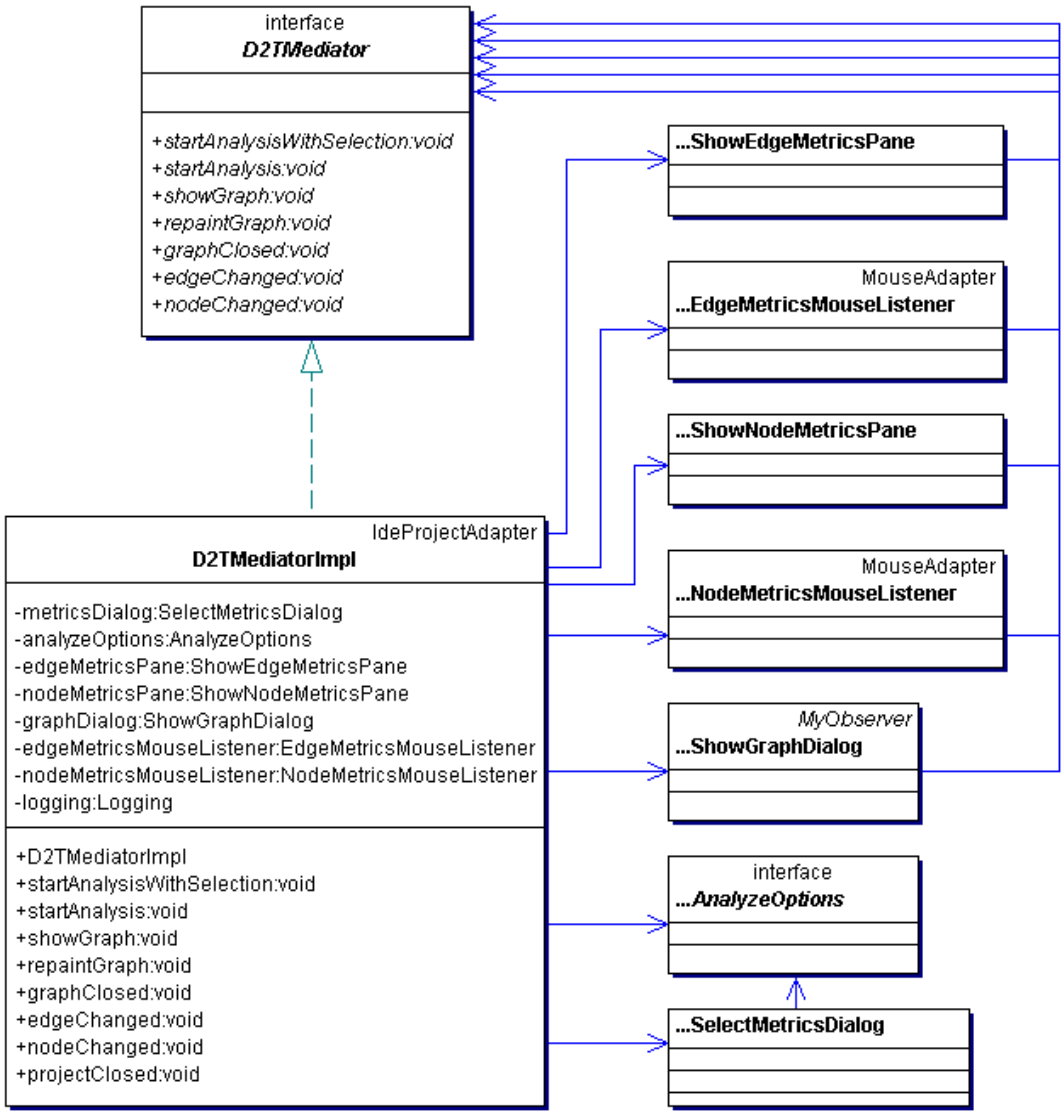


Abbildung 23: Umsetzung des Entwurfsmusters Mediator

Die Verwendung des Entwurfsmusters Mediator bringt folgende Vorteile:

- Logik zur Programmablaufsteuerung wird zentral an einer Stelle gehalten.
- Dadurch geringere Komplexität und besseres Verständnis.
- Ferner erleichterte Wartbarkeit und Erweiterbarkeit der Anwendung.
- Vermeidung von Zyklenbildung durch wechselseitige Abhängigkeiten und insbesondere des rekursiven Aufrufs der Metrikanalyse.

6.8.1 Beispiel für verworfene Entwurfsalternative

Alternativ kam die Einführung einer zentralen Synchronisationskomponente in Betracht, bei der die Registrierung aller betroffenen Komponenten erfolgen hätte müssen. Jede Änderung in der Anzeige wäre dann über den Synchronisationsmanager an alle bei ihm registrierten Komponenten gemeldet worden. Dieser Entwurf stellte sich leider erst in der Implementierungsphase als nicht zielführend heraus, so dass ein Redesign des Entwurfs erforderlich wurde. Die Probleme, die dies notwendig machten, waren bei diesem Entwurf insbesondere:

- Durch die Integration der Anzeigemodule in *Together* war es dem Anwender möglich, die Anzeigemodule wie jede andere originäre *Together*-Schnittstellenkomponente zu schließen. Dieses Schließen der Komponenten konnte jedoch leider nicht im Rahmen des Ereignismanagements beobachtet werden. Dies führte dazu, dass die beim Synchronisationsmanager registrierten Komponenten nicht konsistent gehalten werden konnten. Da der Zustand der geschlossenen Komponenten ferner undefiniert - im Sinne von zufällig - war, führte dies zu unvorhersagbaren Programmabläufen.
- Das Synchronisationsmanagement deckte nur den Teilbereich der konsistenten Anzeige aller Komponenten ab. Der Start der erneuten Metrikanalyse konnte darüber nicht gelöst werden.

6.9 Abschließendes Klassendiagramm des Entwurfs

Nachfolgend das aus dem Entwurf resultierende Klassendiagramm von Design2Test:

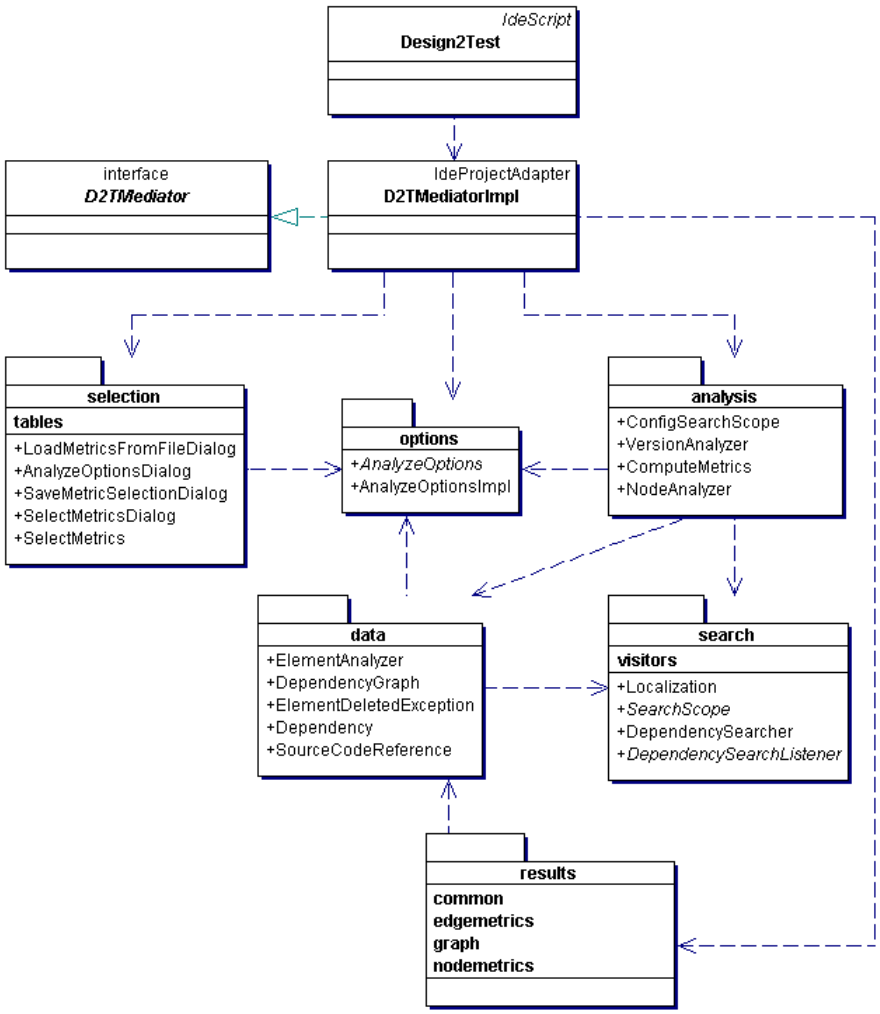


Abbildung 24: Entwurfsklassendiagramm von Design2Test

7 Implementierung

7.1 Einleitung

Begonnen wurde die Implementierung mit Version 5.02 des *Together ControlCenters*. Nachdem sich schon bald zeigte, dass einige Fehler in der Programmierschnittstelle direkten Einfluss auf den Realisierungsaufwand der Integration hatten, wurde bald nach Veröffentlichung des neuesten Releases auf die Version 6.0 gewechselt. Bei der aktuellsten Version von *Together* handelt es sich augenblicklich um die Version 6.01. Leider ist die Programmierschnittstelle trotz einiger Verbesserungen und Erleichterungen auch in diesen Versionen weiterhin an einzelnen Stellen fehlerbehaftet. Dazu mehr in den folgenden Abschnitten, die sich der Implementierung der im Entwurf konzipierten Teilkomponenten widmen. Ferner zeigte sich, dass die Erweiterung von *Together* über das *Open API* eine äußerst zeitaufwändige und umständliche Angelegenheit darstellen kann. Auf diese grundsätzlichen Probleme während der Implementierung wird in einem abschließenden Abschnitt näher eingegangen.

7.2 Erweiterung von *Together* (class Design2Test)

Um ein Modul ausführen zu können, müssen die kompilierten `class`-Dateien unter dem Modulpfad von *Together* (`$TGH$\modules\com\togethersoft\modules`) abgelegt sein (`$TGH$` bezeichnet hierbei das Installationsverzeichnis des *ControlCenters*). *Together* durchsucht sämtliche in diesem Modulpfad abgelegten Verzeichnisse auf das Vorhandensein einer Manifest-Datei, die das Modul und dessen Eigenschaften deklariert. Nachstehend die Manifest-Datei des ausgelieferten Moduls *Design2Test*:

```
Name: Design To Test
Time: Startup
Class-Path: $TGH$\modules\com\togethersoft\modules\design2test
Main-Class: com.togethersoft.modules.design2test.Design2Test
```

Durch Angabe der `Class-Path`-Eigenschaft ist es prinzipiell möglich, von der von *Together* verwendeten Paketstruktur (`com.togethersoft.modules`) abzuweichen. Allerdings führt dies dazu, dass einige der Schnittstellenmethoden des *Together Open API* nicht mehr der Spezifikation entsprechen. So scheitert beispielsweise die Ermittlung des `home`-Verzeichnisses des Moduls durch die Methoden `getModuleHomeDirectory(java.lang.Class)` des Interfaces `IdeManager` sowie der Klasse `ResourceUtil` und müsste ersatzweise relativ aufwändig realisiert werden. Die Realisierung von *Design2Test* erfolgt deshalb ebenfalls auf Basis der Paketstruktur der *Together*-Module. Jedoch ermöglicht der `Class-Path`-Mechanismus zumindest die Positionierung des *ImproveT*-Pakets ohne Anpassungen im Modulverzeichnis.

7.3 Komponente Abhängigkeitssuche (package search)

Um den Entwurf (siehe Abschnitt 6.3) wie beschrieben umzusetzen, ist in der Implementierung vor allem auf folgende Aspekte zu achten:

- Der komplette Sourcecode muss durchlaufen werden.
- Jede relevante Abhängigkeit muss erkannt werden.
- Werden einzelne Codeabschnitte von verschiedenen *Visitoren* durchlaufen, darf keine Abhängigkeit redundant erfasst werden.

- Nicht relevante Abhängigkeiten (z.B. Attribute vom eigenen Klassentyp) müssen ignoriert werden.

Um die vorgenannten Punkte sicherzustellen, wurde parallel zur Implementierung der Suchkomponente ein Referenzprojekt mit allen relevanten Abhängigkeiten aufgebaut. Ferner wurde versucht, auch die übrigen aufgeführten Konstellationen durch eine geeignete Implementierung im Referenzprojekt abzudecken. Anhand dieses Referenzprojekts wird sukzessive die Implementierung der Suchkomponente getestet.

Nachstehend folgt anhand einer relativ einfach zu behandelnden Abhängigkeit eine kurze Schilderung des in den meisten Fällen typischen Vorgehens bei der Identifizierung von relevanten Abhängigkeiten (es handelt sich um einen komprimierten und aufbereiteten Auszug aus dem Originalsourcecode):

```
public Object visitAttribute( SciAttribute sciAttribute ) {
   SciElement referencedElement =
              sciAttribute.getType().getReferencedElement();
   SciClass containingClass = sciAttribute.getContainingClass();

[...]

   // references out of classpath are ignored
   if( referencedElement == null ) {
      return null;
   }

[...]

   // ignoring the Attributes of own Class Type (e.g. singletonInstance)
   if( referencedElement == containingClass ) {
      return null;
   }

   // its an attribute of class type (may also be an java.something.type)
   if( referencedElement instanceof SciClass ) {80

      // add a new Dependency object
      notifyListenerDependencyFound(
              sciAttribute, referencedElement, D_F_TYPE,
              deleteNewLinesAndSpaces(sciAttribute.getDeclarationText()),
              sciAttribute.getType().getPositions());
        }
   // else it is an attribute of primitive type (int, boolean, ...):
   //      => nothing to do

   return null;
}
```

Zusammengefasst lässt sich sagen, dass jede potentielle Abhängigkeit vor der Benachrichtigung des angemeldeten `DependencySearchListener`s auf ihre jeweilige Relevanz geprüft wird. Die Basis für die zu erfolgenden Prüfungen ist dabei ihre Beschreibung in der Übersicht der Abhängigkeiten (Anhang B).

Sämtliche Varianten bei der Ermittlung der Abhängigkeiten an dieser Stelle zu beschreiben, würde den Rahmen dieses Dokuments sprengen. Die nachfolgenden Beispiele beschränken sich deshalb darauf, noch einige interessante Implementierungsteile zu erläutern.

Zunächst ein Blick in die Implementierung der Klasse `myElementVisitor`:

```
public Object visitClass( SciClass sciClass ) {
   // is this class excluded from search?
   if( searchScope != null && !searchScope.isInScope( sciClass)) {
      return null;
   }

   // notify the listener that a class was found (so we can insert all
   // classes as nodes in the graph independently from a dependency)
   notifyListenerClassFound( sciClass);

   // getting all inheritances
   SciInheritance nextInheritance = null;
   SciElement referencedElement = null;
   SciInheritanceEnumeration inheritances = sciClass.inheritances();
   while( inheritances.hasMoreElements() ) {
      nextInheritance = inheritances.nextSciInheritance();
      referencedElement = nextInheritance.getReferencedElement();

      // break if the reference points out of classpath
      if( referencedElement == null )
         break;

      // generally isImplementation works as desired -
      // but for anonymous classes isImplementation is always false!!
      if( nextInheritance.isImplementation() ||
                 (sciClass.hasProperty( SciProperty.UNNAMED) &&
                  referencedElement.hasProperty( SciProperty.INTERFACE)) ){
         // add a new Dependency object
         notifyListenerDependencyFound(
            sciClass, nextInheritance.getReferencedElement(), D_IMPLEM,
            deleteNewLinesAndSpaces(sciClass.getDeclarationText()),
            nextInheritance.getPositions());
      }
[...]
```

Erwähnenswert sind die durch Fettdruck hervorgehobenen Textstellen:

- Die in der Aufgabenstellung vorgesehene Konfiguration der Suche wird durch Übergabe und Auswertung einer Instanz vom Typ `SearchScope` gelöst.
- Während der Implementierungsphase wurde die Anforderungsspezifikation dahingehend präzisiert bzw. erweitert, dass auch Klassen, die nicht an Abhängigkeiten beteiligt sind, in den Abhängigkeitsgraphen eingefügt werden.
- Der Umgang mit anonymen Klassen ist von diversen Schwierigkeiten durch diesbezügliche Unzulänglichkeiten des *Open API* geprägt. Die Berücksichtigung anonymer Klassen führt meist zur Notwendigkeit einer Sonder- bzw. Fehlerbehandlung (*workaround*).

Als nächstes Beispiel wird ein Codeausschnitt aus der Visitor-Klasse `myNewExpressionReferenceVisitor` angeführt. Diese Klasse wurde speziell zur Analyse derjenigen Abhängigkeiten eingeführt, die innerhalb der Verwendung eines `new`-Operators auftreten können:

```
public Object visitReference( SciReference sciReference ) {
   // only the reference-expressions are interesting at this point
   if( !( sciReference instanceof SciReferenceExpression ) )
      return null;

   // the element to which the reference points
   SciElement referencedElement = sciReference.getReferencedElement();

   // references out of classpath are null AND
   // unfortunately references to the implicite standard constructor also!
   if( referencedElement == null) {
      // search alternatively for the class-type itsself!
      SciType type = null;

      // we must look for the SciNewExpression and its SciType
      SciExpression parent =
                  ((SciReferenceExpression)sciReference).getParent();
      while( parent != null) {
         if( parent instanceof SciNewExpression) {
            type = parent.getType();
            if( type != null) {
               referencedElement = type.getReferencedElement();
               if( referencedElement != null) {
                  addNewInstanceDependency( sciReference, referencedElement,
                                            (SciClass)referencedElement);
               }
            }
         }
         parent = parent.getParent() ;
      }

      return null;
      }
[...]
```

Die gekennzeichneten Programmstellen basieren auf dem für Testbarkeitsbetrachtungen unglücklichen Umstand, dass der implizite parameterlose Java-Default-Konstruktor nicht im Programmcode vorhanden ist. Das Einfügen dieses Konstruktors in kompilierte `class`-Dateien wird erst bei der Übersetzung des Programms durch den Compiler vorgenommen. Aufgrund dieser Tatsache ist das *Together Open API* in diesen Fällen nicht in der Lage, eine Referenz auf ein im Programmcode vorhandenes Konstruktor-Element zu finden. Da es sich aus der Sicht der Testbarkeitsanalyse hierbei um eine kritische, weil festverdrahtete, Abhängigkeit handelt, musste dieses Problem umgangen werden. Hierzu wurde sukzessive der Kontext des zu analysierenden Ausdrucks erweitert, bis die zugrunde liegende `SciNewExpression` erreicht wurde. Hier war es dann möglich, den Typ der neu erzeugten Instanz zu bestimmen und zumindest eine Referenz auf die den Typ repräsentierende Klasse zu erhalten.

7.4 Komponente Abhängigkeitsgraph (package data)

Für die Erzeugung der Einträge in den Abhängigkeitsgraphen sind Informationen notwendig, die über den vom *Open API* für die Elemente des Sourcecodes bereit gestellten Funktionsumfang hinausgehen. Diese zusätzliche Funktionalität wird innerhalb der Hilfsklasse `ElementAnalyzer` implementiert:

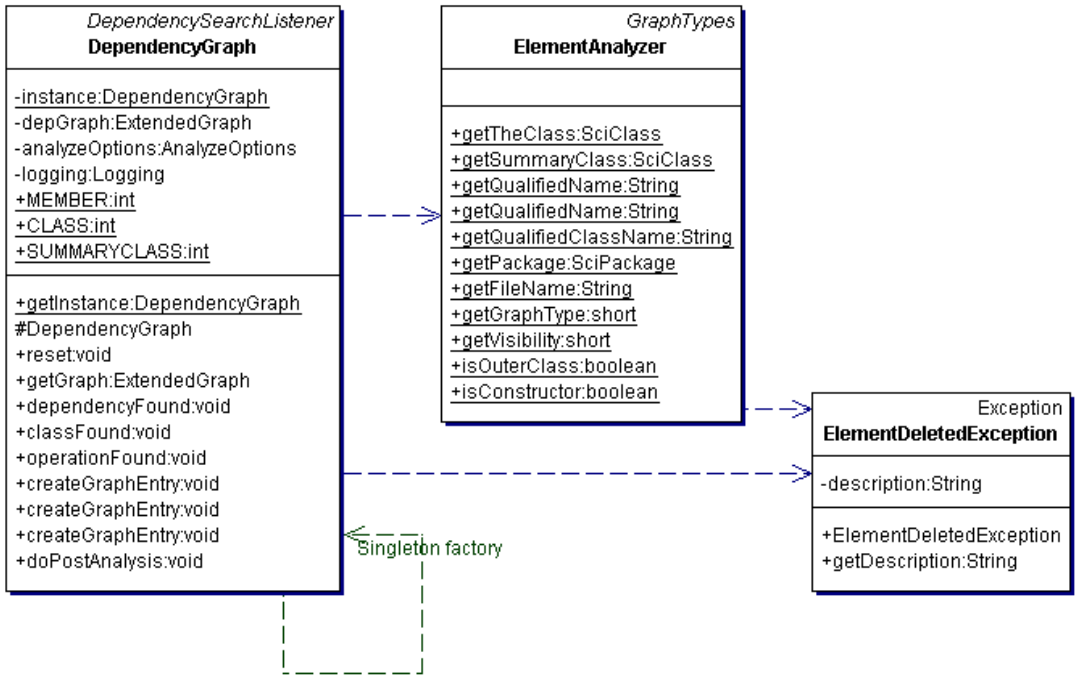


Abbildung 25: Hilfsklassen für die Graphmodellierung

Im Zusammenhang mit anonymen Klassen traten die in Abschnitt 7.8 beschriebenen Probleme auf. Die in diesem Paket realisierte Klasse `ElementDeletedException` dient zur Behandlung der sich aus dem Verlust von Elementen anonymer Klassen ergebenden Probleme.

Ein Schwachpunkt des Entwurfs (siehe Abschnitt 6.4) ist die redundante Speicherung der an der Abhängigkeit beteiligten *Together*-Elemente im Abhängigkeitsgraphen. Neben den Einträgen als Knoten des Graphen, werden diese Elemente als Attribute der Instanzen der Klasse `Dependency` auch als Inhalt der Kanten gespeichert. Es wird zwar derzeit auch über die Attribute der `Dependency`-Instanzen auf die *Together*-Elemente zugegriffen, aber die Implementierung könnte - zwar mit etwas Aufwand aber ohne größere Probleme - dahingehend performanter gestaltet werden, dass der Zugriff ausschließlich über die Knoteneinträge des Graphen erfolgt. Für die Refaktorisierung blieb an dieser Stelle jedoch leider keine Zeit mehr.

7.5 Komponente Analysesteuerung (package analysis)

Eine der Aufgaben der Komponente zur Analysesteuerung umfasst auch den Start der Abhängigkeitssuche. Aus folgendem Programmfragment der Klasse `ComputeMetrics` wird nochmals die Vorgehensweise bei der Benutzung der Suchkomponente deutlich:

```
// recurse through the model and searches for dependencies
DependencySearcher dependencySearcher = new DependencySearcher();
dependencySearcher.setDependencySearchListener( DependencyGraph.getInstance());
dependencySearcher.setSearchScope( new ConfigSearchScope());
dependencySearcher.search();
```

Nach der Erzeugung der Suchkomponente wird zunächst der erzeugte Abhängigkeitsgraph zur Benachrichtigung über die gefundenen Abhängigkeiten angemeldet. Nachdem die vom Anwender eingegebenen Suchoptionen gesetzt wurden, wird die Suche nach Abhängigkeiten gestartet.

7.6 Komponente Metrikauswahl (package selection)

Um das Modul unabhängig von Änderungen hinsichtlich der in *ImproveT* verfügbaren Metriken zu halten, wurde für den Zugriff auf die implementierten Testbarkeitsmetriken der *Reflection*-Mechanismus von Java verwendet. Als Basis für den Zugriff dienen Konfigurationsdateien, in denen jene Metriken eingetragen werden, die für die Testbarkeitsanalyse verfügbar sein sollen. Damit ist beispielsweise beim Hinzukommen neuer Metriken eine Neukompilierung des *Design2Test*-Moduls nicht erforderlich. Bei der Auswahl der zu berechnenden Metriken stellt sich die Implementierung des Konstruktors beispielsweise wie folgt dar:

```
public EdgeMetricsTableModel() {
   String moduleHome =
         IdeAccess.getIdeManager().getModuleHomeDirectory( this.getClass());
   try {
      BufferedReader edgeMetricsIn =
           new BufferedReader( new FileReader( moduleHome +
               Localization.getString( "ImproveT.EdgeMetricsInputFile")));

      String metricSetName = null;
      Class metricCls = null;
      Object metricInst = null;
      for(;;) {
         metricSetName = edgeMetricsIn.readLine();
         if( metricSetName == null)
            break;

         try {
            // get the metric class
            metricCls = Class.forName( Localization.getString(
                           "ImproveT.EdgeMetricPackage") + metricSetName);
            metricInst = metricCls.newInstance();
            if( ((MetricSet)metricInst).hasPercentageValue())
               ((MetricSet)metricInst).setAsPercentage( true);
            metricSets.add( (MetricSet)metricInst);
         } catch( ClassNotFoundException e) {
            IdeMessageManagerAccess.printMessage(IdeMessageType.ERROR_MODAL,
               "MetricClass not found: " + e.toString() );
         } catch( InstantiationException e) {
            IdeMessageManagerAccess.printMessage(IdeMessageType.ERROR_MODAL,
               "MetricClass cannot be instantiated: " + e.toString() );
         } catch( IllegalAccessException e) {
            IdeMessageManagerAccess.printMessage(IdeMessageType.ERROR_MODAL,
               "MetricClass cannot be accessed: " + e.toString() );
      }
   }
[...]
```

Die Angabe der Metriken in den Konfigurationsdateien erfolgt durch Eintrag einer Zeile mit dem Namen der die Metrik realisierenden Klasse. Die Implementierung bei der Übergabe der Metriken an die Analysemodule von *ImproveT* erfolgt analog.

7.7 Ausgabekomponenten (package results)

Durch die Realisierung der Koordination und Synchronisation der einzelnen Teilkomponenten von *Design2Test* nach dem Entwurfsmuster Mediator wurde der Implementierung dieser Funktionalität ein beträchtlicher Teil der Komplexität genommen. Die Implementierung der einzelnen

Benutzeroberflächen erfordert zwar den üblichen Aufwand ist aber ebenfalls ohne größere Schwierigkeiten zu leisten.

Bei der Integration der Anzeigekomponenten in das Together ControlCenter ist allerdings darauf zu achten, dass die Nebenläufigkeit der in der Ausgabeschicht von *Together* auszuführenden Prozesse gewahrt wird. Hierfür stellt Together in seiner IDE-Schicht (siehe Abschnitt 5.4.1) über das Interface `IdeManager` folgende Methoden zur Verfügung:

```
public void addCommandAndWait(Runnable command)
public void addCommandToQueue(Runnable command)
public void addCommandToQueue(Runnable command, String name)
```

Eine einfache Implementierung einer Dialoganzeige sieht beispielsweise wie folgt aus:

```
JMenuItem graphItem = new JMenuItem( "Graph");
graphItem.addActionListener( new ActionListener() {
...public void actionPerformed( ActionEvent e) {
......IdeAccess.getIdeManager().addCommandToQueue(new Runnable() {
.........public void run() {
............// open graph and highlight current edge
............mediator.showGraph();
............mediator.edgeChanged( edge, table);
.........}
......});
...}
});
```

Für die Implementierung der Druckausgaben wurden die bereits existierenden Komponenten des *Quality Assurance*-Moduls (*QA*-Modul) wiederverwendet. Dies erspart an dieser Stelle erheblichen Aufwand, der bei einer eigenen Implementierung zu leisten gewesen wäre. Ein kleiner Nachteil dieser Variante ist allerdings, dass die Druckausgaben von *Design2Test* damit nur bei geladenem *QA*-Modul verfügbar sind. Da dieses Modul aber bei der Standardinstallation des Together *ControlCenters* automatisch geladen wird und damit üblicherweise für den auf Qualitätssicherung bedachten Anwender verfügbar sein wird, kann dieser Nachteil in Kauf genommen werden.

7.8 Komponente Modulkoordination (class D2TMediator)

Die Komponente zur Modulkoordination steuert als Mediator das Zusammenspiel aller Komponenten von *Design2Test*. Bei der zentralen Methode zur Steuerung des Programmablaufs handelt es sich um `startAnalysis()`:

```
public void startAnalysis() {
   IdeAccess.getIdeManager().addCommandToQueue(
      new Runnable() {
         public void run() {
            ComputeMetrics computeMetrics =
                    new ComputeMetrics( metricsDialog.getEdgeMetricsClassNames(),
                                        metricsDialog.getNodeMetricsClassNames(),
                                        analyzeOptions, logging);
            computeMetrics.start();

            // now show the results
            edgeMetricsMouseListener =
                    new EdgeMetricsMouseListener( D2TMediatorImpl.this);
            edgeMetricsPane =
                    new ShowEdgeMetricsPane( D2TMediatorImpl.this,
                                             computeMetrics.getVersionAnalysis(),
                                             edgeMetricsMouseListener);
            edgeMetricsPane.openPane();

            nodeMetricsMouseListener =
                    new NodeMetricsMouseListener( D2TMediatorImpl.this);
            nodeMetricsPane =
                    new ShowNodeMetricsPane( D2TMediatorImpl.this,
                                             computeMetrics.getNodeAnalysis(),
                                             nodeMetricsMouseListener);
            nodeMetricsPane.openPane();

            if( graphDialog != null) {
               graphDialog.closeDialog();
               showGraph();
            }
         }
      });
}
```

Man sieht wie zunächst die Abhängigkeitssuche und Metrikberechnung gestartet wird und im Anschluss daran die Anzeigekomponenten bei ihrer Erzeugung mit den Ergebnissen versorgt und zur Anzeige gebracht werden. Die Kontrolle über alle Komponenten kann so beim Mediator verbleiben.

7.9 Probleme der Implementierungsphase

Bei der Entwicklung von *Together*-Modulen zeigen sich einige grundsätzliche Erschwernisse:

- Die einzige Möglichkeit ein Modul zu debuggen, besteht darin eine zweite Instanz des *ControlCenters* im Debugger der ersten Instanz zu starten. Ein System mit 512 MB Arbeitsspeicher und 800 MHz- Prozessor benötigt für den Start dieser zweiten Instanz ca. 4 Minuten.
- Die komplexeste Teilkomponente, das Durchlaufen des Sourcecodes und das Erkennen der Abhängigkeiten, kann nur sinnvoll auf Grundlage eines real verfügbaren *Together*-Projekts unter Verwendung des *Together Open API* getestet werden. Dieser Teilbereich kann also nicht unabhängig von einer gestarteten Instanz des *Together ControlCenter* getestet werden, so dass das Debug-Problem voll zum Tragen kommt.
- Die Dokumentation der Programmierschnittstelle ist in vielen Teilen sehr knapp gehalten. Es gibt zwar einige einfache Beispielmodule, die mit dem *ControlCenter* ausgeliefert werden, doch beim tieferen Einstieg in die Modulprogrammierung steht als einziges Hilfsmittel die nicht immer besonders aussagekräftige *javadoc*-Kommentierung des *Open API* zur Verfügung.

Im Nutzer-Forum der Firma *TogetherSoft* (www.togethercommunity.com) finden sich einige kritische Beiträge zu diesen Problematiken. Hierbei ist jedoch zu bedenken, dass der Fokus des *Together*-Entwicklungsteams sicher auf der eigentlichen Funktionalität von *Together* und nicht auf der Unterstützung der Programmierschnittstelle liegt.

An verschiedenen Stellen der Implementierung traten auch noch andere Unzulänglichkeiten des *Together Open API* zu Tage.

So mussten beispielsweise in der Version 5.02 von *Together* anonyme Klassen aus der die Klasse erzeugenden `new`-Anweisung noch manuell extrahiert werden. Mit der Version 6.0 wurde das Interface `SciJavaHelper` allerdings um eine Methode `anonymousClasses(SciClass)` mit der gewünschten Funktionalität erweitert.

Diese Änderung führte aber vermutlich dazu, dass ab der Version 6.0 innerhalb von Methoden der Rumpf von anonymen Klassen beim Durchlaufen des Methodenrumpfes ebenfalls durchlaufen wurde. Dies war in der Version 5.02 noch nicht der Fall, was rein intuitiv die korrekte und für die Zwecke dieser Arbeit in jedem Falle die wesentlich vorteilhaftere Art der Realisierung war. Leider blieb keine Wahl und es musste ein umständlicher *workaround* eingefügt werden, der die in den Methoden ebenfalls erkannten Abhängigkeiten anonymer Klassen ausfiltert. Aus der Dokumentation lässt sich im Übrigen nicht ablesen, welche der beiden beschriebenen Realisierungen aus Sicht der *Together*-Entwickler beabsichtigt ist.

Im Zusammenhang mit der Behandlung anonymer Klassen besitzt *Together* leider auch noch die unangenehme Eigenschaft, dass die Referenzen auf die Modellelemente, die für die Repräsentation anonymer Klassen erzeugt und durch das *Open API* geliefert werden, während der Laufzeit des Programms verloren gehen, so dass der versuchte Zugriff zu einer `NullPointerException` führt. In einer ersten Version des Moduls wurde beispielsweise der Abhängigkeitsgraph erst nach dem vollständigen Durchlaufen des Sourcecodes des zu analysierenden Projekts erstellt. Aufgrund des Programmfehlers kam es bei dieser Variante relativ unvorhersehbar (die Größe des analysierten Projekts spielte dabei eine Rolle) zu den angesprochenen `NullPointerExceptions` und zu der zu Beginn dieses Abschnitts bereits beschriebenen äußerst aufwändigen Fehlersuche in *Together*.

In der abschließend realisierten Version wird der Abhängigkeitsgraph deshalb bereits während des Durchlaufs durch die *Together*-Repräsentation des Sourcecodes und der Ermittlung der Abhängigkeiten erzeugt. Dies führt zu keinerlei Nachteilen. Um den Fehler ferner bei der Anzeige der Testbarkeitsmetriken und der dort angebotenen Möglichkeit zur Navigation zum Sourcecode ebenfalls zu umgehen, wurde die Implementierung der Klasse `SourceCodeReference` um ein zusätzliches Attribut `String sourceCodeFileName` erweitert. Dadurch kann auch an dieser Stelle der Zugriff auf das nicht mehr referenzierte Element zur Bestimmung des Dateinamens vermieden werden.

7.10 Fazit

Im Hinblick auf die Komplexität der Realisierung lag der eindeutige Schwerpunkt auf der Erkennung und Ermittlung der im Vorfeld festgelegten Abhängigkeiten über das *Together Open API*. Bei der Analyse bestimmter Programmteile und der Erkennung einzelner Abhängigkeiten ergaben sich teilweise nur mit erheblichem Aufwand in den Griff zu bekommende Probleme.

Ein weiterer aufwändiger Teil der Implementierung, der mehrfach einen Rückschritt in die Entwurfsphase und eine Überarbeitung des Entwurfs erforderte, war die Synchronisation und Verknüpfung der Ausgabekomponenten. Hier ergab sich erst durch die Verwendung eines geeigneten Design-Patterns eine Reduzierung der in diesem Zusammenhang aufgetretenen Probleme.

Teil III - Fallbeispiel

Im dritten Teil dieser Arbeit erfolgt die Verwendung von *Design2Test* zur Analyse eines Softwaresystems auf Probleme hinsichtlich seiner Testbarkeit. Nach einer nochmaligen Motivation der Aufgabenstellung und der Beschreibung des zu analysierenden Softwaresystems, folgt die Analyse und Bewertung von als testkritisch identifizierten Abhängigkeiten des Systems und eine ausführliche Diskussion der hinter den Testproblemen verborgenen Entwurfsmängel. Den Abschluss des zweiten Teils bildet die konkrete Entfernung verschiedener Entwurfsprobleme und Abhängigkeiten.

8 Einführung

8.1 Motivation

Die Motivation der in dieser Arbeit realisierten Erweiterung des *Together ControlCenters* (kurz: *Together*) um die Funktionalität von *ImproveT* war, den Nutzern der *Together*-Entwicklungsumgebung ein Hilfsmittel zur Einschätzung und insbesondere Verbesserung der Testbarkeit ihrer Anwendung zu geben. Der Entwickler erhält durch die integrierte Testbarkeitsanalyse für alle Abhängigkeiten seines Systems die in Abschnitt 1.3 vorgestellten Testbarkeitsmetriken geliefert, angereichert um weitere Informationen, die ihm zusätzliche Hinweise zur Bewertung testkritischer Abhängigkeiten geben.

Für den zweiten Teil dieser Diplomarbeit sah die Aufgabenstellung vor, den entwickelten Ansatz anhand eines Beispiels auf seine Effektivität zu untersuchen (siehe Aufgabenstellung in Kapitel 2). Im Rahmen der Analyse eines real existierenden Softwaresystems sollten insbesondere folgende Fragen näher untersucht und soweit möglich beantwortet werden:

- Bieten die Testbarkeitsmetriken die Möglichkeit zur Identifikation jener Programmteile, die die Testbarkeit eines objektorientierten Softwaresystems übermäßig verschlechtern?
- Welche der verfügbaren Testbarkeitsmetriken und ergänzenden Informationen zeigen dies in welchen Konstellationen und Ausprägungen an?
- Wie hoch ist der Aufwand, der zur Beseitigung von testkritischen Abhängigkeiten geleistet werden muss?
- In welchem Maße und Umfang trägt die Beseitigung der Abhängigkeiten zu einer Verbesserung der Testbarkeit bei?

Für die Beantwortung dieser Fragen waren folgende Arbeitsschritte vorgesehen:

- Ermittlung der Testbarkeitsmetriken für das im Fallbeispiel zu untersuchende Softwaresystem und Bewertung der Ergebnisse.
- Exemplarische Durchführung von Refaktorisierungsschritten zur Beseitigung von ausgewählten Abhängigkeiten und Bestimmung des Aufwands.
- Erstellung und Durchführung von geeigneten Testfällen für das System vor und nach der Entfernung von ausgewählten Abhängigkeiten und Bestimmung der Testaufwände.

- Sofern möglich, Entwicklung und Beschreibung von Heuristiken zur Identifizierung von vergleichsweise einfach zu entfernenden testkritischen Abhängigkeiten.

Nach einer kurzen Beschreibung des zu untersuchenden Softwaresystems (*SeminarIS*) im nächsten Abschnitt, beginnt die Analyse von *SeminarIS* mit der Ermittlung der Testbarkeitsmetriken. Darauf aufbauend erfolgt die Bewertung jener Abhängigkeiten in *SeminarIS*, deren Beitrag zur Zahl der durchschnittlichen direkten und indirekten Abhängigkeiten einer *SeminarIS*-Komponente (Klasse) am höchsten ist (Kapitel 9).

Im Rahmen dieser Bewertung wird sich zeigen, dass *SeminarIS* unter einigen massiven Testproblemen leidet, die auf problematische und zum Teil auch fehlerhafte Entwurfsentscheidungen sowie einzelne Implementierungsfehler zurückzuführen sind:

- Beinahe alle in der Datenhaltungsschicht angesiedelten Klassen sind in einem einzigen Abhängigkeitszyklus enthalten.
- Große Systemteile sind von der verwendeten Datenbank bzw. dem eingesetzten Persistenz-Rahmenwerk statisch abhängig.
- Bei der Implementierung erfolgte mehrfach eine Verletzung der Drei-Schichten-Architektur durch Zugriff aus tieferliegenden Schichten auf eine der übergeordneten Schichten, was zu immensen zyklischen Abhängigkeiten in *SeminarIS* führte.

Diese Erkenntnisse wurden beim Versuch, erste Testfälle für *SeminarIS* zu erstellen, bestätigt. Hier zeigte sich, dass bei der Durchführung von Klassentests (*Unit Tests)* die Idealvorstellung des isolierten Tests einer Klasse nicht einmal ansatzweise realisiert werden konnte. So wird beispielsweise auch beim Test der meisten Benutzerschnittstellenklassen stets die Datenbank gestartet, bereits abgefragt und unter Umständen durch das Persistenz-Rahmenwerk zusätzlich benötigter Code generiert und kompiliert (Proxyklassen). Dieser Mechanismus ließe sich nur durch Veränderung des Originalcodes verhindern, gängige Strategien zur Herstellung isolierter Bedingungen greifen hingegen nicht (z.B. Einsatz von Teststellvertretern).

Aufgrund dieser Erfahrungen erschien es sinnvoll und hilfreich, abweichend von der ursprünglich geplanten Vorgehensweise, eine ausführliche Untersuchung dieser Entwurfsprobleme an die Bewertung der testkritischen Abhängigkeiten anzuschließen und die Testfallerstellung vorerst zurückzustellen. Ziel der Untersuchung in Kapitel 10 war, einen Überblick zu gewinnen, an welchen Stellen und in welchem Umfang diese grundlegenden Entwurfsprobleme die Testbarkeit des Systems berühren.

Als Ergebnis dieser Untersuchung musste erkannt werden, dass die durch die angezeigten Testprobleme erkannten Entwurfsprobleme so weitreichend sind, dass eine Entfernung einzelner Abhängigkeiten kein erfolgversprechendes Vorgehen bei der Refaktorisierung des Systems darstellt. Die Refaktorisierung zielt folglich auf die generelle Beseitigung der beschriebenen Entwurfsprobleme ab. Die Beseitigung dieser Probleme einschließlich der Diskussion alternativer Vorgehensweisen bei der Refaktorisierung wird in Kapitel 11 beschrieben.

Die in der Aufgabenstellung vorgesehene Bestimmung von Testaufwänden vor und nach der Refaktorisierung, konnte aus Zeitgründen nicht mehr durchgeführt werden. Insbesondere deshalb, weil die in *SeminarIS* vor der Refaktorisierung vorhandenen Testprobleme zu einem außerordentlich hohen Aufwand bei der Erledigung der Testaufgaben geführt hätten.

8.2 Beschreibung des zu analysierenden Softwaresystems

Als Fallbeispiel bot sich die Analyse eines Projekts an, an dessen Erstellung der Verfasser dieser Arbeit im Rahmen eines Software-Praktikums am Lehrgebiet Praktische Informatik III der FernUniversität Hagen im Sommersemester 2001 beteiligt war. Bei diesem Software-Projekt handelt es sich um ein Seminar-Informationssystem (*SeminarIS*), das eine fiktive Schulungsfirma bei der Planung und Durchführung von Seminaren sowie der Verwaltung ihrer Kundendaten unterstützen soll. An der Entwicklung des Projekts waren insgesamt acht Studenten der FernUniversität beteiligt. Der gesamte Entwicklungsaufwand dürfte (grob geschätzt) bei etwa 9 Personenmonaten gelegen haben.

Die Realisierung erfolgte auf Basis des Kurses *Software Engineering II – Objektorientierte Softwareentwicklung* der FernUniversität Hagen **[**SWEII99**]**. Als Programmiersprache war Java für die Implementierung vorgegeben. Die Realisierung erfolgte entsprechend der Vorgabe in einer Drei-Schicht-Architektur, bestehend aus den Schichten „Benutzerschnittstelle“, „Anwendungslogik“ und „Datenhaltung“. Als Datenbanksystem fand die zum damaligen Zeitpunkt als Open-Source verfügbare relationale Datenbank *InstantDB* (http://instantdb.tripod.com) Verwendung. Die Verbindung zur Datenbank wurde über ein Persistenz-Rahmenwerk (*PersistenzRW*) realisiert, das im Rahmen einer Diplomarbeit an der FernUniversität entwickelt wurde. Als Entwicklungsumgebungen standen den Studenten in der Implementierungsphase wahlweise *Together* oder *JBuilder* der Firma *Borland* zur Verfügung.

Der Schwerpunkt des Software-Praktikums lag aufgrund der begrenzten Zeit auf der Entwurfs- und Implementierungsphase, so dass die Durchführung von Tests überwiegend unsystematisch und insbesondere undokumentiert erfolgte. Für die Analyse des Projekts hinsichtlich seiner Testbarkeit stehen demzufolge keinerlei Unterlagen aus der Entwicklungsphase, wie beispielsweise Testfälle, zur Verfügung.

9 Testbarkeitsanalyse ausgewählter Abhängigkeiten

9.1 Einleitung

Design2Test liefert im Rahmen der Testbarkeitsanalyse von *SeminarIS* Testbarkeitsmetriken für 273 Komponenten (Klassen) und 1872 Abhängigkeiten zwischen diesen Klassen. Bezogen auf die in Abschnitt 1.3 vorgestellten Testbarkeitsmetriken (erweitert um eine Metrikvariante ACDh, bei deren Errechnung nur statische Abhängigkeiten berücksichtigt werden), ergeben sich für *SeminarIS* folgenden Werte:

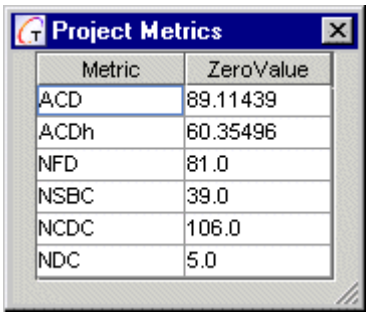


Abbildung 26: Projektmetriken SeminarIS

Jede *SeminarIS*-Klasse ist im Durchschnitt von ca. 89 anderen Klassen direkt oder indirekt syntaktisch abhängig (Metrik ACD). Dabei handelt es sich bei etwa zwei Drittel dieser Abhängigkeiten um statische Abhängigkeiten (Metrik ACDh). Der Abhängigkeitsgraph von SeminarIS enthält 5 Zyklen (Metrik NDC), an denen 106 Klassen beteiligt sind (Metrik NCDC). Es existieren 81 Abhängigkeiten, deren Entfernung einen Beitrag zum Aufbrechen der Zyklen leisten würde (Metrik NFD). Möchte man im Testprozess die vorhandenen Zyklen aufbrechen, so sind 39 Teststellvertreter (Stubs) hierfür notwendig (Metrik NSBC).

Aufgrund der Vielzahl der Informationen, die von *ImproveT* geliefert werden, ist es in dieser Arbeit nicht möglich, eine vollständige Analyse und Überprüfung aller Ergebnisse durchzuführen. Wir konzentrieren uns auf jene Abhängigkeiten, deren ersatzlose Entfernung den stärksten Beitrag zur Reduzierung der Abhängigkeit von Komponenten (gemessen an der Metrik ACD) leisten würde.

Die nachfolgende Tabelle gibt einen Überblick über die 30 Abhängigkeiten mit den höchsten rACD-Werten in *SeminarIS*:

Abhängige Klasse	Funktionalität bereitstellende Klasse	dep_comp	is_feedb	is_intpk	rACD	rACDh	sub_hw	subedges	Typ der Abhängigkeit
Dozentenv ereinbarung	DVKostenBerechnenK	1	0	1	8,5	9,9	1	9	create and access
SeminarisK	SeminarisDatenbank	1	0	1	6,0	12,6	1	5	static access
SVAendernErfassenAA	Seminarty pAusw aehlenAA	0	0	1	2,2	3,3	1	2	create and access
DruckDokumentDK	SeminarisK	0	0	1	2,1	2,9	4	4	static access
SVAendernErfassenAA	LeitungsauftragAusw aehlenAA	1	0	1	1,8	3,0	1	3	create and access
Seminarty pAendernErfassenAA	DozentenvereinbarungAusw aehlenAA	0	0	1	1,6	2,3	1	3	create and access
LeitungsauftragAendernErfassenAA	SVAusw aehlenAA	1	1	1	1,6	2,5	1	2	create and access
DVKostenBerechnenK	DozentenvereinbarungLoeschenK	1	1	0	1,6	1,9	1	1	static access
SeminarbelegungAA	SVAusw aehlenAA	0	0	1	1,6	2,4	2	3	create and access
SeminarisH	SachbearbeiterA	1	1	0	1,5	2,4	1	3	static access
Erw eitertPersistent	SeminarisDatenbank	1	0	1	1,4	4,4	1	1	static access
SVKurzS	Seminarv eranstaltung	0	0	1	1,1	-	0	8	dep. to abstract class
MeldungsAnzeigeA	ExceptionNachrichtA	0	0	0	1,0	0,8	1	3	create and access
TeilnahmeFirmenSVRelation	TeilnahmeFirmenSVPaar	1	0	0	1,0	1,2	2	2	create and access
SVSeminarty pRelation	SVSeminarty pPaar	1	0	0	1,0	1,2	2	2	create and access
SVAnsprechpartnerRelation	SVAnsprechpartnerPaar	1	0	0	1,0	1,2	2	2	create and access
RechnungsempfaengerRelation	RechnungsempfaengerPaar	1	0	0	1,0	1,2	2	2	create and access
SeminarisK	LogProtokollDateiA	1	1	1	1,0	1,2	1	2	create and access
SeminarisK	DebugProtokollDateiA	1	1	1	1,0	1,2	1	2	create and access
AnstellungRelation	AnstellungPaar	1	0	0	1,0	1,2	2	2	create and access
BelegungRelation	BelegungTripel	1	0	0	1,0	1,2	2	2	create and access
BuchtRelation	BuchtPaar	1	0	0	1,0	1,2	2	2	create and access
FirmenAnsprechpartnerRelation	FirmenAnsprechpartnerPaar	1	0	0	1,0	1,2	2	2	create and access
LeitungsauftragRelation	LeitungsauftragTripel	1	0	0	1,0	1,2	3	3	create and access
DozentenvereinbarungRelation	DozentenvereinbarungAnfragen	1	0	0	1,0	1,2	6	7	static access
DozentenvereinbarungTripel	DVSchluessel	1	0	0	1,0	0,8	5	7	create and access
Seminarv eranstaltungOrdner	OeffentlicheSeminarv eranstaltung	1	1	1	0,8	1,1	1	1	static access
DozentenvereinbarungLoeschenK	IDozentenvereinbarungK	0	0	0	0,8	1,0	1	1	static access
Seminarty p	Seminarty pOrdner	1	0	1	0,8	1,0	2	4	static access

Abbildung 27: SeminarIS-Abhängigkeiten mit dem höchsten rACD-Wert

ImproveT ist in der Lage, über die in Abschnitt 1.3 vorgestellten Testbarkeits- und Reduktionsmetriken hinaus, zur genaueren Einordnung der Ergebnisse eine Vielzahl weiterer Informationen zu Abhängigkeiten und Komponenten zu liefern. Im Rahmen dieser Arbeit erfolgt eine Beschränkung auf einige wenige dieser Kennziffern. In den Darstellungen der Abhängigkeitsmetriken sind dies Informationen

- ob die betreffende Abhängigkeit in einem Zyklus enthalten ist (dep_comp),
- es sich um eine Kante handelt, die einen Beitrag zur Auflösung eines Zyklus leisten kann (is_feedb),
- ob die Abhängigkeit innerhalb desselben Paketes verläuft (is_intpk),
- an wie vielen Stellen im Sourcecode die Abhängigkeit ausgelöst wird (subedges),
- wie viele dieser auslösenden Codefragmente statischer Natur sind (sub_hw) und
- welcher Abhängigkeitstyp aus der Gesamtmenge der Abhängigkeiten, die konkrete Auslöser einer Abhängigkeit zwischen Klassen sind, aus Testbarkeitssicht der kritischste ist (d_type).

Darüber hinaus existieren eine Vielzahl von Metriken, die zu allen im System vorhandenen Komponenten ergänzende Aussagen in Bezug auf die von ihnen ausgehenden Abhängigkeiten liefern. Hier existieren Kennziffern über die

- Anzahl der von ihnen ausgehenden direkten und indirekten Abhängigkeiten zu anderen Komponenten innerhalb des Systems (cd, cd_tr),
- wie viele dieser Abhängigkeiten statischer Natur sind (cd_h..., cd_s...),
- bei wie vielen es sich um Typabhängigkeiten handelt (cd_t...),
- wie viele dieser Abhängigkeiten in der öffentlichen Schnittstelle sichtbar sind und wie viele aus den Methodenrümpfen heraus verlaufen (cd_ci..., cd_im...).
- Daneben existieren für die Testbarkeitsmetrik ACD zwei weitere, über alle ein- und ausgehenden Abhängigkeiten aggregierte Varianten auf Klassenebene (rACD_in, rACD_out).

In den nachfolgenden Abschnitten werden die durch *ImproveT* verfügbaren Informationen für eine Untersuchung und Bewertung der 30 Abhängigkeiten mit den höchsten rACD-Werten verwendet.

9.2 Top rACD-Abhängigkeiten

Die Analyse der Abhängigkeiten ist jeweils in zwei Abschnitte unterteilt. Im ersten Teil werden die Abhängigkeitsmetriken und die für die an der Abhängigkeit beteiligten Klassen ermittelten Komponentenmetriken angegeben. Bei den ersten Abhängigkeiten werden die Metriken zum Verständnis ihrer Bedeutung ausführlich erläutert, während bei den restlichen Abhängigkeiten nur noch Auffälligkeiten bei den Metriken beschrieben werden und sie sonst unkommentiert bleiben.

Im zweiten Teil erfolgt eine Bewertung der Abhängigkeiten. Diese soll die Entstehung und die Ursachen der Abhängigkeit aufzeigen, eine Einschätzung ihres Einflusses auf die Testbarkeit vornehmen und gegebenenfalls die Möglichkeiten zur Entfernung der Abhängigkeit beleuchten. Auch diese Bewertung erfolgt bei den ersten Abhängigkeiten etwas ausführlicher. In der Folgezeit beschränkt sich die Bewertung aus Zeit- und Platzgründen darauf, in aller Kürze auf die wichtigsten Aspekte hinzuweisen.

9.2.1 Dozentenvereinbarung - DVKostenBerechnenK

9.2.1.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	1	8,5	9,9	6,2	0,0	2,8	-20,0	1	9	create and access

Abbildung 28: Abhängigkeitsmetriken Dozentenvereinbarung - DVKostenBerechnenK

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	14	7	5	0	2	7	1	0	6	1	3	2	9	3	5	4	89	65	62	9,4	8,8
B	6	1	1	0	0	5	0	0	5	0	2	3	2	1	4	4	89	65	62	8,8	7,8

Abbildung 29: Komponentenmetriken Dozentenvereinbarung (A) und DVKostenBerechnenK (B)

Erläuterungen zu den Abhängigkeitsmetriken:

Die Abhängigkeit ist Teil eines Zyklus (dep_comp) und verläuft über Paketgrenzen hinweg (is_intpk). Wir werden später sehen, dass die beiden Pakete in unterschiedlichen Anwendungsschichten angesiedelt sind und die Abhängigkeit durch ihre Richtung die Hierarchie verletzt (Abschnitt 10.3 - *Verletzung der Drei-Schicht-Architektur*).

Der mit Abstand höchste rACD-Wert des gesamten Projekts stimmt in etwa mit dem rACDh-Wert überein. Das Entfernen der Abhängigkeit würde die Anzahl der Zyklen von 5 auf 6 erhöhen (rNDC). Dies geschieht infolge der Zersplittung eines größeren Zyklus in zwei kleinere. Da gleichzeitig die Anzahl der Zyklenkomponenten von 106 auf 103 sinken würde (rNCDC), wird insbesondere auch die durchschnittliche Zyklengröße von ca. 21 auf ca. 17 Komponenten verringert.

Die Abhängigkeit wird an einer Stelle durch Erzeugung eines Objekts ausgelöst, auf das dann mehrfach zugegriffen wird (d_type, sub_hw und subedges).

Erläuterungen zu den Komponentenmetriken:

Die Klasse Dozentenvereinbarung hängt von insgesamt 14 anderen Komponenten ab (cd). Ihr rACD_out-Wert liegt unwesentlich höher als der rACD-Wert der Abhängigkeit. Die übrigen ausgehenden Abhängigkeiten haben demzufolge keinen größeren Einfluss auf den ACD-Wert des Systems.

Die Anzahl der direkten und indirekten Abhängigkeiten (cd_tr) beider Klassen resultiert unter anderem aus der Beteiligung an einem Zyklus. Die Anzahl statischer Abhängigkeiten liegt in beiden Klassen im Mittelfeld, der höchste Wert für die Metrik cd_h beträgt 15. Die Klasse Dozentenvereinbarung besitzt immerhin zur Hälfte Typabhängigkeiten (cd_t), die zu einem Großteil zu Interfaces führen (cd_t_if).

Welche Auswirkung eine hohe Anzahl von direkten und indirekten statischen Abhängigkeiten (Metrik cd_h_tr) auf den Test einer Klasse haben kann, soll beispielhaft für alle untersuchten Abhängigkeiten nachfolgend an der Klasse `Dozentenvereinbarung` veranschaulicht werden. Der Metrik cd_h_tr zufolge benötigt der Modultest von `Dozentenvereinbarung` ohne Änderungen an der Implementierung statischen Zugriff auf wenigstens 65 andere Klassen bei der Durchführung von Testfällen. Um die Auswirkung auf die Testdurchführung auf einfache Weise zu illustrieren, wurde in der Klasse `Dozentenvereinbarung` ein Testtreiber zum Erzeugen einer Dozentenvereinbarung implementiert.

Dies bedeutet den Test einer einzigen, vergleichsweise einfachen Methode. Nachfolgende Abbildung veranschaulicht die in *SeminarIS* vorhandenen Abhängigkeiten durch Auflistung der beim Test geladenen Klassen. Es handelt sich um die mit der Option `-verbose` von der virtuellen Javamaschine gewonnene Information, die um alle Einträge von Klassen bereinigt wurde, die nicht zum Paket *SeminarIS* gehören:

```
[Loaded DatenhaltungP.IErweitertPersistent]
[Loaded DatenhaltungP.ErweitertPersistent]
[Loaded DatenhaltungP.SeminarisP.IDozentenvereinbarung]
[Loaded DatenhaltungP.SeminarisP.Dozentenvereinbarung]
[Loaded ExterneSystemeP.PersistenzP.SeminarisDatenbank]
[Loaded UtilP.ExceptionP.SeminarisException]
[Loaded UtilP.ExceptionP.SchluesselVorhandenException]
[Loaded AnwendungslogikP.SeminarisP.SeminarisK]
[Loaded ExterneSystemeP.KonfigurationP.Konfiguration]
[Loaded UtilP.ExceptionP.KonfigurationException]
[Loaded UtilP.SynchronisationP.SynchronisationsManager]
[Loaded ExterneSystemeP.ProtokollP.ProtokollManager]
[Loaded ExterneSystemeP.ProtokollP.IProtokollAusgabeKunde]
[Loaded ExterneSystemeP.ProtokollP.ProtokollDateiA]
[Loaded ExterneSystemeP.ProtokollP.DebugProtokollDateiA]
[Loaded ExterneSystemeP.ProtokollP.LogProtokollDateiA]
[Loaded UtilP.ExceptionP.CommitOrRollbackWithoutStartTransactionException]
[Loaded DatenhaltungP.SeminarisP.IGeschaeftspartner]
[Loaded DatenhaltungP.SeminarisP.Geschaeftspartner]
[Loaded UtilP.ExceptionP.UngueltigerSchluesselException]
[Loaded DatenhaltungP.SeminarisP.IFirma]
[Loaded DatenhaltungP.SeminarisP.Firma]
[Loaded DatenhaltungP.SeminarisP.IPerson]
[Loaded DatenhaltungP.SeminarisP.Person]
[Loaded DatenhaltungP.SeminarisP.IBelegung]
[Loaded DatenhaltungP.SeminarisP.Belegung]
[Loaded UtilP.ExceptionP.AssoziationsException]
[Loaded DatenhaltungP.SeminarisP.IGeld]
[Loaded DatenhaltungP.SeminarisP.ISeminarveranstaltung]
[Loaded DatenhaltungP.SeminarisP.Seminarveranstaltung]
[Loaded DatenhaltungP.SeminarisP.IFirmenSeminarveranstaltung]
[Loaded DatenhaltungP.SeminarisP.FirmenSeminarveranstaltung]
[Loaded DatenhaltungP.SeminarisP.ILeitungsauftrag]
[Loaded DatenhaltungP.SeminarisP.Leitungsauftrag]
[Loaded DatenhaltungP.TransformationenP.DozentenvereinbarungRelation]
[Loaded DatenhaltungP.TransformationenP.DozentenvereinbarungAnfragen]
[Loaded DatenhaltungP.SeminarisP.Dozentenvereinbarung$1]
[Loaded DatenhaltungP.SeminarisP.Dozentenvereinbarung$2]
[Loaded DatenhaltungP.SeminarisP.Dozentenvereinbarung$3]
[Loaded DatenhaltungP.SeminarisP.Dozentenvereinbarung$4]
[Loaded DatenhaltungP.SeminarisP.IOeffentlicheSeminarveranstaltung]
[Loaded DatenhaltungP.SeminarisP.ISeminartyp]
[Loaded DatenhaltungP.SeminarisP.Geld]
[Loaded DatenhaltungP.SeminarisP.Seminartyp]
[Loaded DatenhaltungP.TransformationenP.IDozentenvereinbarungTripel]
[Loaded DatenhaltungP.TransformationenP.DozentenvereinbarungTripel]
[Loaded DatenhaltungP.TransformationenP.DVSchluessel]
```

Abbildung 30: Geladene Klassen beim Erzeugen einer Dozentenvereinbarung (insg. 47)

Zusätzlich werden 71 Klassen des Persistenz-Rahmenwerks und 50 Klassen der Java-Datenbank geladen. Dies zeigt den gewaltigen Aufwand, der für einen einzigen relativ simplen Methodentest alleine von der Laufzeitumgebung betrieben werden muss.

9.2.1.2 Bewertung der Abhängigkeit

Bei `Dozentenvereinbarung` handelt es sich um eine in der Anforderungsspezifikation definierte Assoziationsklasse. Eine Dozentenvereinbarung wird mit einer Person (Dozent) über einen von ihm durchgeführten Seminartyp abgeschlossen. Bei der Klasse `DVKostenBerechnenK` handelt es sich um eine Kontrollklasse in der Logikschicht. Sie bietet die Funktionalität zur Berechnung der Kosten, die der Einsatz eines Dozenten für eine Seminarveranstaltung verursachen würde.

Der Ursprung dieser Klasse lässt sich auf die Anforderungsspezifikation zurückführen. In der textuellen Klassenspezifikation der Klasse `Dozentenvereinbarung` wird folgende Methode definiert:

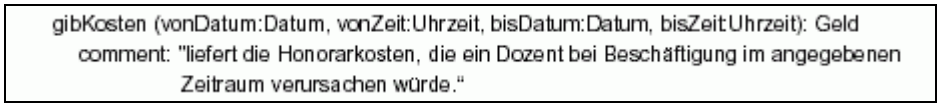

gibKosten (vonDatum:Datum, vonZeit:Uhrzeit, bisDatum:Datum, bisZeit:Uhrzeit): Geld
comment: "liefert die Honorarkosten, die ein Dozent bei Beschäftigung im angegebenen Zeitraum verursachen würde."

Abbildung 31: Auszug aus der textuellen Klassenspezifikation der Klasse Dozentenvereinbarung

Vermutlich war diese Methode zur Verwendung für den Anwendungsfall *Seminarveranstaltung kalkulieren* vorgesehen. Dort wird sie jedoch in dieser Form nicht benötigt. Während in den anderen Entwicklungsgruppen des Praktikums diese Methode aufgrund der ungenauen Spezifikation (z.B. fehlen täglicher Seminarbeginn und Seminarende für eine exakte Berechnung) und ihrer Entbehrlichkeit nicht realisiert wurde, resultierte im vorliegenden Projekt daraus die Klasse `DVKostenBerechnenK`.

Problematisch an dieser Abhängigkeit ist insbesondere, dass sie von der tiefer liegenden Datenhaltungsschicht in die höher liegende Kontrollschicht verläuft (siehe Abschnitt 10.3 - *Verletzung der Drei-Schicht-Architektur*). Vermutlich resultiert dies aus einem anderen Entwurfsproblem von *SeminarIS*, der Realisierung der Domänenklassen als Fassadenklassen, die einen Teil der von Ihnen zu leistenden Aufgaben an andere Klassen delegieren (siehe Abschnitt 10.7 - *Realisierung der Domänenklassen als Fassaden*). Die Fassadenfunktionalität der Klasse `Dozentenvereinbarung` verführte möglicherweise in diesem Fall dazu, eine Delegation der Aufgabe an die Kontrollschicht vorzunehmen.

Die fachliche Anforderung des Anwendungsfalles *Seminarveranstaltung kalkulieren* konnte ohne die Funktionalität der Klasse `DVKostenBerechnenK` realisiert werden. Ein weiterer Zugriff auf die Klasse erfolgt lediglich aus einem deaktivierten Programmteil. Somit kann die Klasse `DVKostenBerechnenK` und damit die hier diskutierte Abhängigkeit ersatzlos entfallen. Der Aufwand hierfür ist sehr gering (siehe Abschnitt 11.2 - *Realisierung einer strikten Drei-Schichten-Architektur*).

Nach der Refaktorisierung kann in der Klasse `Dozentenvereinbarung` der Test der abhängigkeitsauslösenden und entfernten Methode `gibKosten()` entfallen. Durch den Wegfall der Klasse `DVKostenBerechnenK` und der ausschließlich von ihr genutzten Klasse `Zeitpunkt` ergibt sich eine weitere Reduzierung des Testaufwandes.

9.2.2 SeminarisK - SeminarisDatenbank

9.2.2.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	1	6,0	12,6	2,5	7,7	0,9	-40,0	1	5	static access

Abbildung 32: Abhängigkeitsmetriken SeminarisK - SeminarisDatenbank

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	0	0	0	0	6	0	0	6	0	2	4	6	6	0	0	89	65	62	8,8	8,2
B	12	5	4	0	1	7	0	0	7	1	0	6	2	2	10	5	89	65	62	17,7	16,8

Abbildung 33: Komponentenmetriken SeminarisK (A) und SeminarisDatenbank (B)

Erläuterungen zu den Abhängigkeitsmetriken:

Die Abhängigkeit ist Teil eines Zyklus (dep_comp) und verläuft über Paketgrenzen hinweg (is_intpk). Der im Vergleich zum rACD-Wert doppelt so hohe rACDh-Wert fällt auf.

Das Entfernen der Abhängigkeit würde die Anzahl der Zyklen von 5 auf 7 erhöhen (rNDC). Dies geschieht infolge der Zersplittung eines größeren Zyklus in zwei kleinere, da derzeit durch die Abhängigkeiten von `SeminarisDatenbank` zu einzelnen Domänenklassen (`Belegung`, `Leitungsauftrag`, `Geschaeftspartner`, ...) eigentlich disjunkte Zyklen in der Datenhaltungsschicht zu einem einzigen größeren Zyklus verschmelzen:

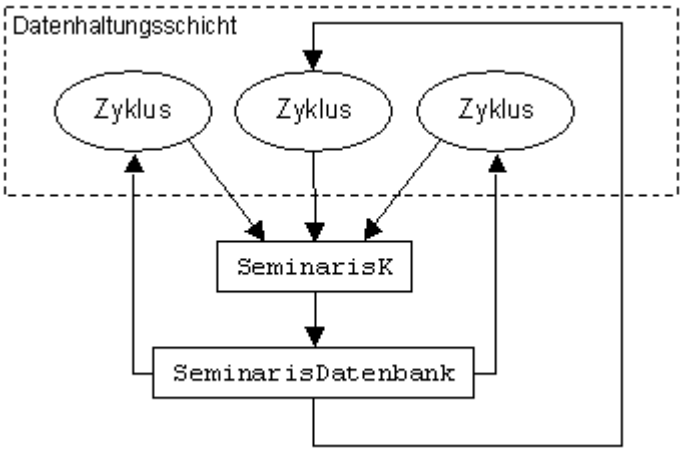


Abbildung 34: Zyklen in Datenhaltungsschicht

Die Anzahl der Zyklenkomponenten bleibt mit 105 nahezu unverändert (rNCDC), wodurch die durchschnittliche Zyklengröße von ca. 21 auf ca. 15 Komponenten verringert wird.

Aus Abbildung 34 ergibt sich ferner eine Erklärung für den hohen rACD-Wert. Es handelt sich hier um eine sogenannte *Hub*-Abhängigkeit: Es wird von einer großen Anzahl von Klassen (komplette Kontrollschicht und einzelne Datenhaltungsklassen) über `SeminarisK` auf die Klasse `SeminarisDatenbank` zugegriffen. Von dort erfolgt der Zugriff auf die zyklisch verbundene Datenhaltungsschicht. Über die hier diskutierte Abhängigkeit erfolgt demnach die Verknüpfung von Subgraphen, woraus sich der Begriff *Hub* (Mittelpunkt, Netzknoten) ableitet.

Ausgelöst wird die Abhängigkeit an einer Stelle durch statischen Zugriff auf die einzige Datenbankinstanz, auf deren Methoden dann mehrfach nicht-statisch zugegriffen wird (d_type, sub_hw und subedges).

Erläuterungen zu den Komponentenmetriken:

Auffällig sind insbesondere die extrem hohen Werte der Metriken rACD_in und rACD_out für die Klasse `SeminarisDatenbank`. Auf die Ursachen wird in der Bewertung näher eingegangen.

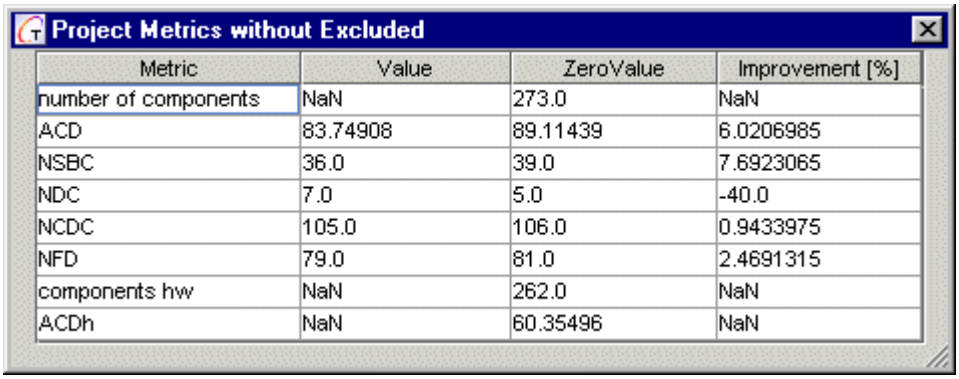
9.2.2.2 Bewertung

Bei `SeminarisK` handelt es sich um die Haupt-Kontrollklasse der Anwendung. Sie stellt verschiedene globale Attribute für die Logikschicht zur Verfügung und liefert auf Anfrage für bestimmte Domänen eindeutige Bezeichner. Die Klasse `SeminarisDatenbank` ist von der Datenbank-Klasse des verwendeten Persistenz-Rahmenwerkes abgeleitet. Dabei wurden die Transaktionsmethoden der Superklasse überschrieben und Funktionalität zur Initialisierung der in `SeminarisK` verwalteten eindeutigen Bezeichner hinzugefügt.

Die Abhängigkeit basiert demnach auf folgenden Punkten:

- In `SeminarisK` wird versucht, den Klassen der Kontrollschicht zentralen Zugriff auf die Basiskomponenten der Anwendung zu ermöglichen. Dies ist ein grundsätzliches Problem, auf das in Abschnitt 10.6 *Zugriff auf globale Basiskomponenten* näher eingegangen wird.
- Ferner wurde in `SeminarisK` und `SeminarisDatenbank` die Verwaltung von eindeutigen Bezeichnern für einzelne Domänen realisiert. Dies führt insbesondere zur Abhängigkeit der Basiskomponente `SeminarisDatenbank` von der Datenhaltungsschicht. Diese Problematik wird in Abschnitt 10.5 *Zugriff auf Datenhaltung* näher beleuchtet.

Für eine exaktere Bestimmung des Einflusses der genannten Probleme auf den rACD-Wert der Abhängigkeit stehen in der *Together*-Erweiterung weitere Funktionalitäten zur Verfügung. Zur Erinnerung nochmals die Veränderungen bei den Projektmetriken durch Entfernung der Abhängigkeit:



Metric	Value	ZeroValue	Improvement [%]
number of components	NaN	273.0	NaN
ACD	83.74908	89.11439	6.0206985
NSBC	36.0	39.0	7.6923065
NDC	7.0	5.0	-40.0
NCDC	105.0	106.0	0.9433975
NFD	79.0	81.0	2.4691315
components hw	NaN	262.0	NaN
ACDh	NaN	60.35496	NaN

Abbildung 35: Projektmetriken nach Entfernung der Ursprungsabhängigkeit

Alternativ wird nun die Abhängigkeit von `SeminarisK` zu `SeminarisDatenbank` beibehalten und stattdessen werden die Abhängigkeiten von `SeminarisDatenbank` zur Datenhaltungsschicht entfernt. Hierbei ist bemerkenswert, dass alle entfernten Abhängigkeiten einen rACD-Wert von 0 aufweisen und darüber hinaus tatsächlich erst die vollständige Entfernung aller acht Abhängigkeiten überhaupt eine Veränderung der Metriken bewirkt. Mit anderen Worten bleibt bei der Entfernung jeder echten Teilmenge dieser acht Abhängigkeiten der ACD-Wert des Systems also unverändert (Maskierungseffekt):

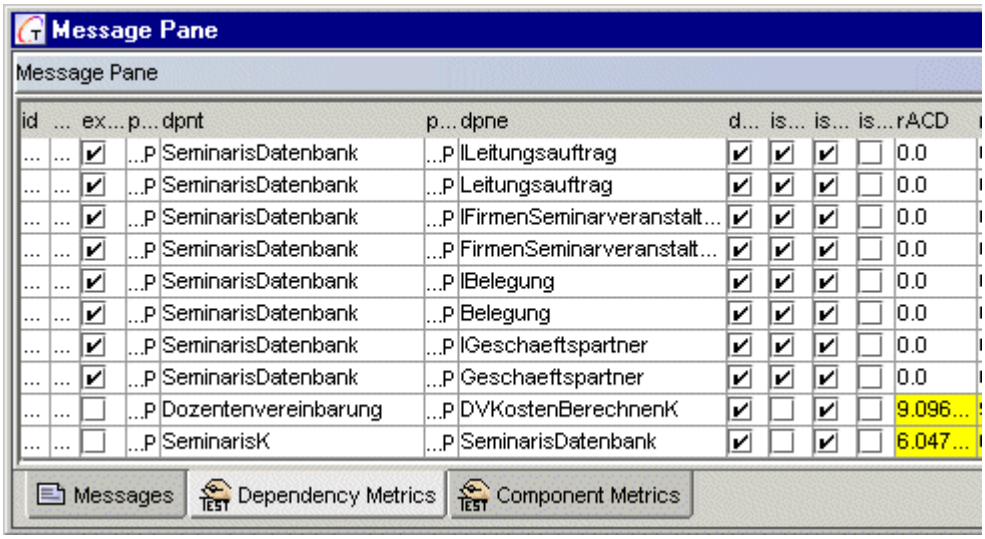
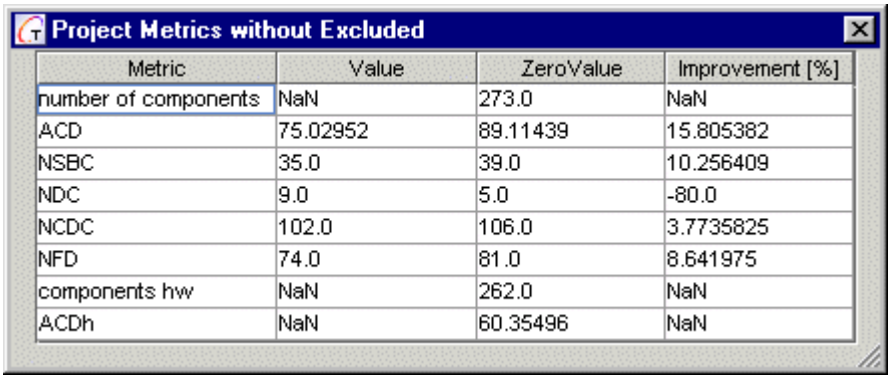

Message Pane
Message Pane
id ... ex... p... dpnt p... dpne d... is... is... is... rACD
... ...P SeminarisDatenbank ...P ILeitungsauftrag 0.0
... ...P SeminarisDatenbank ...P Leitungsauftrag 0.0
... ...P SeminarisDatenbank ...P IFirmenSeminarveranstalt... 0.0
... ...P SeminarisDatenbank ...P FirmenSeminarveranstalt... 0.0
... ...P SeminarisDatenbank ...P IBelegung 0.0
... ...P SeminarisDatenbank ...P Belegung 0.0
... ...P SeminarisDatenbank ...P IGeschaeftspartner 0.0
... ...P SeminarisDatenbank ...P Geschaeftspartner 0.0
... ...P Dozentenvereinbarung ...P DVKostenBerechnenK 9.096...
... ...P SeminarisK ...P SeminarisDatenbank 6.047...
Messages
Dependency Metrics
Component Metrics

Abbildung 36: Entfernung der Abhängigkeiten von der Datenbank zur Datenhaltung

Nach Entfernung dieser Abhängigkeiten, ergeben sich folgende Projektmetriken:



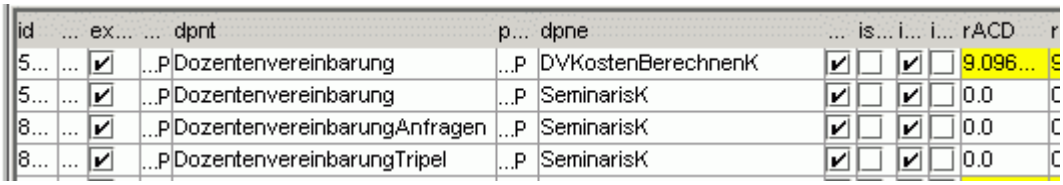
Metric	Value	ZeroValue	Improvement [%]
number of components	NaN	273.0	NaN
ACD	75.02952	89.11439	15.805382
NSBC	35.0	39.0	10.256409
NDC	9.0	5.0	-80.0
NCDC	102.0	106.0	3.7735825
NFD	74.0	81.0	8.641975
components hw	NaN	262.0	NaN
ACDh	NaN	60.35496	NaN

Abbildung 37: Projektmetriken nach Entfernung der Abhängigkeiten zur Datenhaltung

Durch die Verlagerung der oben beschriebenen Methoden zur Verwaltung von eindeutigen Domänenbezeichnern, würde also eine Verringerung des ACD-Wertes um 15,8 Prozent erreicht.

Dies lässt die Schlussfolgerung zu, dass der hohe rACD-Wert der untersuchten Abhängigkeit im Wesentlichen durch die Abhängigkeiten von `SeminarisDatenbank` in die Datenhaltungsschicht ausgelöst wird. Dies wird dadurch bestätigt, dass die Entfernung der Abhängigkeit von `SeminarisK` nach `SeminarisDatenbank` zum jetzigen Zeitpunkt nur noch eine marginale Veränderung der Metrikwerte (ACD: 74,9) ergibt.

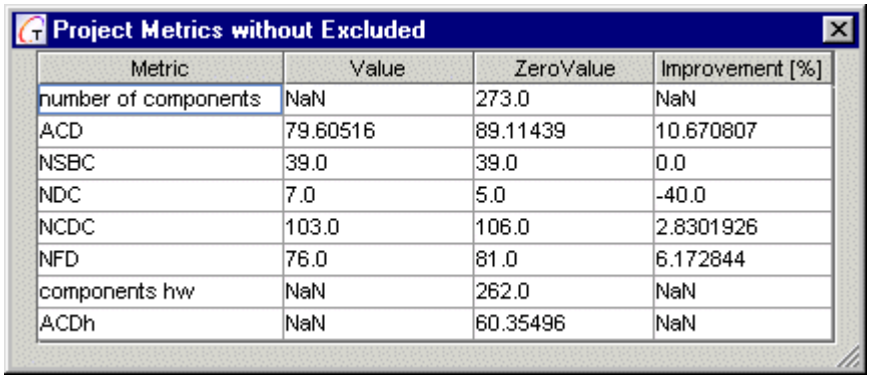
Einfluss auf die Metrikwerte könnten möglicherweise auch noch jene Abhängigkeiten aus der Datenhaltungsschicht in die Kontrollschicht haben (siehe Abschnitt 10.3 - *Verletzung der Drei-Schicht-Architektur*), die zur Erweiterung des Datenhaltungszyklus beitragen (siehe Abbildung 34). Mit der nachstehenden Entfernung der genannten Abhängigkeiten sind sämtliche Abhängigkeiten aus der Datenhaltung in die Anwendungslogik entfernt. Bemerkenswert ist, dass die Entfernung ohne Hinzufügen zusätzlicher Abhängigkeiten geschehen kann (siehe Abschnitt 11.2 - *Realisierung einer strikten Drei-Schichten-Architektur*):



id	...	ex...	...	dpnt	p...	dpne	...	is...	i...	i...	rACD	r
5...	...	✔	...P	Dozentenvereinbarung	...P	DVKostenBerechnenK	✔	☐	✔	☐	9.096...	9
5...	...	✔	...P	Dozentenvereinbarung	...P	SeminarisK	✔	☐	✔	☐	0.0	0
8...	...	✔	...P	DozentenvereinbarungAnfragen	...P	SeminarisK	✔	☐	✔	☐	0.0	0
8...	...	✔	...P	DozentenvereinbarungTripel	...P	SeminarisK	✔	☐	✔	☐	0.0	0

Abbildung 38: Entfernung der Abhängigkeiten aus der Datenhaltung in die Kontrollschicht

Die Entfernung der Abhängigkeiten führt zu folgenden Projektmetriken:



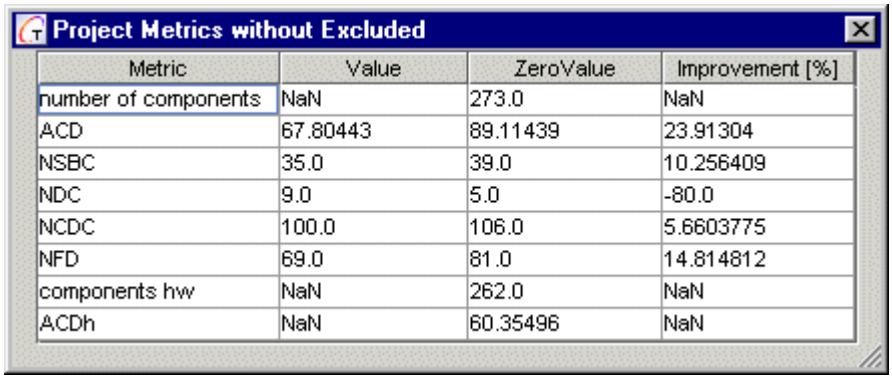
Project Metrics without Excluded

Metric	Value	ZeroValue	Improvement [%]
number of components	NaN	273.0	NaN
ACD	79.60516	89.11439	10.670807
NSBC	39.0	39.0	0.0
NDC	7.0	5.0	-40.0
NCDC	103.0	106.0	2.8301926
NFD	76.0	81.0	6.172844
components hw	NaN	262.0	NaN
ACDh	NaN	60.35496	NaN

Abbildung 39: Projektmetriken ohne Abhängigkeiten aus Datenhaltung in Kontrollschicht

Auch hier kommt es zu einer deutlichen Verringerung des `ACD`-Wertes (11,3Prozent).

Führt man beide Refaktorisierungsschritte zusammen durch, entfernt also die Abhängigkeiten aus `SeminarisDatenbank` in die Datenhaltung und aus der Datenhaltung in die Kontrollschicht, so erhält man sogar folgende Verbesserung der Projektmetriken:



Project Metrics without Excluded

Metric	Value	ZeroValue	Improvement [%]
number of components	NaN	273.0	NaN
ACD	67.80443	89.11439	23.91304
NSBC	35.0	39.0	10.256409
NDC	9.0	5.0	-80.0
NCDC	100.0	106.0	5.6603775
NFD	69.0	81.0	14.814812
components hw	NaN	262.0	NaN
ACDh	NaN	60.35496	NaN

Abbildung 40: Projektmetriken ohne zyklische Abhängigkeiten zwischen Datenhaltung und Anwendungslogik

Diese bisherigen (sehr einfach zu realisierenden) Refaktorisierungsschritte bewirken also bereits eine Reduzierung des `ACD`-Wertes um nahezu ein Viertel. Entfernt man abschließend noch zur Kontrolle die Ausgangsabhängigkeit von `SeminarisK` nach `SeminarisDatenbank`, so stellt man fest, dass die Projektmetriken nahezu unverändert bleiben.

Zusammengefasst lässt sich festhalten, dass der hohe rACD-Wert der hier diskutierten Abhängigkeit auf die Abhängigkeiten von `SeminarisDatenbank` in die Datenhaltung sowie aus der Datenhaltungsschicht in die Kontrollschicht zurückgeführt werden kann. Die hier diskutierte Abhängigkeit war ein hervorragender Indikator für zwei grundlegende Entwurfsprobleme. Bei den erkannten testkritischen Abhängigkeiten handelt es sich um direkt benachbarte (adjazente) Abhängigkeiten der hier diskutierten Abhängigkeit im Abhängigkeitsgraphen.

Auch wenn im Hinblick auf den ACD-Wert des Projekts eine Entfernung der Abhängigkeit `SeminarisK` nach `SeminarisDatenbank` keine nennenswerte Verbesserung mehr bringt, sollte auch sie entfernt werden. Wie in Abschnitt 10.6 - *Zugriff auf globale Basiskomponenten* - beschrieben, handelt es sich auch hier um ein gravierendes Testproblem, da sonst insbesondere keine Konfigurierbarkeit der Datenbank in der Testumgebung möglich ist. Hier erfolgt die Anzeige des Entwurfsproblems unmittelbar durch die Abhängigkeit.

9.2.3 SVAendernErfassenAA - SeminartypAuswaehlenAA

9.2.3.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	1	2,2	3,3	0,0	0,0	0,0	0,0	1	2	create and access

Abbildung 41: Abhängigkeitsmetriken SVAendernErfassenAA - SeminartypAuswaehlenAA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	14	5	0	4	1	9	0	1	8	0	5	3	7	3	7	6	184	166	153	3,8	4,5
B	11	1	0	1	0	10	0	1	9	0	5	4	3	2	8	8	119	95	91	2,3	2,7

Abbildung 42: Komponentenmetriken SVAendernErfassenAA (A) und SeminartypAuswaehlenAA (B)

Erläuterungen zu den Abhängigkeitsmetriken:

Hier handelt es sich um eine Abhängigkeit, die nicht Teil eines Zyklus (dep_comp) ist. Das Entfernen der Abhängigkeit hätte folglich keinerlei Auswirkungen auf Zyklen.

Erläuterungen zu den Komponentenmetriken:

Beide Klassen besitzen eine verhältnismäßig hohe Zahl von ausgehenden Abhängigkeiten, auch wenn der höchste cd-Wert aller Komponenten sogar bei 26 liegt. Der rACD_out-Wert von `SVAendernErfassenAA` liegt doppelt so hoch als der rACD-Wert der Abhängigkeit. Dies ergibt sich durch die zwei Abschnitte tiefer noch zu betrachtende testkritische Abhängigkeit der Klasse zu `LeitungsauftragAuswaehlenAA`.

Die extrem hohe Anzahl der direkten und indirekten Abhängigkeiten (cd_tr) der Klasse `SVAendernErfassenAA` resultiert daraus, dass die Klasse an mehreren Zyklen beteiligt ist. Die Anzahl statischer Abhängigkeiten liegt in beiden Klassen verhältnismäßig hoch (der systemweit höchste Wert für die Metrik cd_h beträgt 15).

9.2.3.2 Bewertung

In der Anforderungsspezifikation heißt es im *main flow* zum Anwendungsfall *Seminarveranstaltung erfassen*:

> Ein Seminarsachbearbeiter **wählt zunächst einen Seminartyp aus**, legt eine neue Seminarveranstaltung (von diesem Seminartyp) an, erfasst die zusätzlichen Angaben und erfasst zumindest einen Leitungsauftrag (extension point: "LA erfassen").

Die in der Anforderungsspezifikation vorgeschriebene Auswahl eines Seminartyps wurde beim Entwurf von *SeminarIS* derart realisiert, dass hierdurch eine Abhängigkeit dieses Anwendungsfalles mit den Anwendungsfällen zur Bearbeitung eines Seminartyps entstand (siehe Abschnitt 10.8). Dies führt zur Abhängigkeit der zu den Anwendungsfällen implementierten Schnittstellenklassen und den damit verbundenen Testproblemen.

9.2.4 DruckDokumentDK - SeminarisK

9.2.4.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	1	2,1	2,9	0,0	0,0	0,0	0,0	4	4	static access

Abbildung 43: Abhängigkeitsmetriken DruckDokumentDK - SeminarisK

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	3	2	0	0	2	1	0	0	1	0	0	1	0	0	3	1	92	66	63	1,8	2,2
B	6	0	0	0	0	6	0	0	6	0	2	4	6	6	0	0	89	65	62	8,8	8,2

Abbildung 44: Komponentenmetriken DruckDokumentDK (A) und SeminarisK (B)

Erläuterungen zu den Abhängigkeitsmetriken:

Hier handelt es sich wiederum um eine Abhängigkeit, die nicht Teil eines Zyklus (dep_comp) ist. Sie verläuft jedoch ebenfalls über Paketgrenzen hinweg (is_intpk).

Der rACD-Wert stimmt in etwa mit dem rACDh-Wert überein. Das Entfernen der Abhängigkeit hätte keinerlei Auswirkungen auf Zyklen. Es handelt sich im übrigen wieder um eine typische *Hub*-Abhängigkeit: Von der Klasse `DruckDokumentDK` leiten sich einige Druckerklassen ab, die über `SeminarisK` und `SeminarisDatenbank` (siehe Abschnitt 9.2.2) indirekte Abhängigkeiten zur Datenhaltung erfahren.

Die Abhängigkeit wird durch mehrfachen Zugriff auf ein statisches Attribut der Klasse SeminarisK ausgelöst (d_type, sub_hw und subedges).

Erläuterungen zu den Komponentenmetriken:

Beide Klassen besitzen nur eine geringe Zahl von ausgehenden Abhängigkeiten. Der rACD_out-Wert von SVAendernErfassenAA liegt etwa beim rACD-Wert der Abhängigkeit.

Die Anzahl der direkten und indirekten Abhängigkeiten (cd_tr) wird bestimmt durch indirekte Abhängigkeit vom Zyklus in der Datenhaltung.

9.2.4.2 Bewertung

Die Abhängigkeit wird ausgelöst durch den statischen Zugriff auf das Attribut `Config` der Klasse `SeminarisK`. Das Attribut liefert die einzige Instanz von `Konfiguration`, die benötigt wird, um die Druckereigenschaften aus der *SeminarIS*-Konfigurationsdatei zu lesen. Durch die Zusammenfassung globaler Attribute in der Klasse `SeminarisK` entsteht in diesem Fall eine indirekte Abhängigkeit der Klasse `DruckDokumentDK` zur Datenbank und in die Datenhaltung. Eine detaillierte Diskussion des Zugriffs auf Basiskomponenten und ein Vorschlag zur Lösung des Problems erfolgt in Abschnitt 10.6 - *Zugriff auf globale Basiskomponenten*.

Problematisch an der vorliegenden Abhängigkeit ist insbesondere, dass aus einem außerhalb der Drei-Schicht-Architektur realisierten Basispaket auf die Kontrollschicht der Mehrschicht-Architektur zugegriffen wird. Hierdurch verliert die Druckkomponente seine Eigenständigkeit. Eine weitere Diskussion erfolgt in Abschnitt 10.4 - *Zugriff auf Kontrollschicht* .

9.2.5 SVAendernErfassenAA - LeitungsauftragAuswaehlenAA

9.2.5.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	1	1,8	3,0	13,6	2,6	11,3	20,0	1	3	create and access

Abbildung 45: Abhängigkeitsmetriken SVAendernErfassenAA - LeitungsauftragAuswaehlenAA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	14	5	0	4	1	9	0	1	8	0	5	3	7	3	7	6	184	166	153	3,8	4,5
B	12	2	0	2	0	10	0	1	9	0	5	4	3	2	9	8	184	166	153	1,8	2,5

Abbildung 46: Komponentenmetriken SVAendernErfassenAA (A) und LeitungsauftragAuswaehlenAA (B)

Erläuterungen zu den Abhängigkeitsmetriken:

Im Gegensatz zur Abhängigkeit 9.2.3 ist diese Abhängigkeit Teil eines Zyklus (dep_comp). Sie verläuft jedoch ebenfalls über Paketgrenzen hinweg (is_intpk).

Der rACD-Wert liegt deutlich niedriger als der rACDh-Wert. Das Entfernen der Abhängigkeit würde die Anzahl der Zyklen von 5 auf 4 verringern (rNDC), bei gleichzeitiger Verringerung der Anzahl der Zyklenkomponenten von 106 auf 94 (rNCDC).

Die Abhängigkeit wird durch Erzeugung eines Objekts ausgelöst, auf das dann mehrfach zugegriffen wird (d_type, sub_hw und subedges).

Erläuterungen zu den Komponentenmetriken:

Beide Klassen besitzen eine verhältnismäßig hohe Zahl von ausgehenden Abhängigkeiten. Der rACD_out-Wert von `SVAendernErfassenAA` liegt doppelt so hoch als der rACD-Wert der Abhängigkeit. Dies ergibt sich durch die zwei Abschnitte höher bereits betrachtete Abhängigkeit zur Klasse `SeminartypAuswaehlenAA`.

Die extrem hohe Anzahl der direkten und indirekten Abhängigkeiten (cd_tr) beider Klassen resultiert unter anderem aus der Zyklusbeteiligung. Die Anzahl statischer Abhängigkeiten liegt in beiden Klassen verhältnismäßig hoch.

9.2.5.2 Bewertung

In der Anforderungsspezifikation heißt es im *main flow* zum Anwendungsfall *Seminarveranstaltung erfassen*:

> Ein Seminarsachbearbeiter wählt zunächst einen Seminartyp aus, legt eine neue Seminarveranstaltung (von diesem Seminartyp) an, erfasst die zusätzlichen Angaben und **erfasst zumindest einen Leitungsauftrag** (extension point: "LA erfassen").

Die in der Anforderungsspezifikation vorgeschriebene Erfassung eines Leitungsauftrags wurde beim Entwurf von *SeminarIS* durch direkte Verzweigung in den zugehörigen Anwendungsfall *Leitungsauftrag erfassen* realisiert. Dies führt zur Abhängigkeit der zu den Anwendungsfällen implementierten Schnittstellenklassen und den damit verbundenen Testproblemen.

Da in diesem Fall darüber hinaus auch eine Abhängigkeit vom Anwendungsfall *Leitungsauftrag erfassen* zum Anwendungsfall *Seminarveranstaltung erfassen* besteht, führt dies sogar zu einer zyklischen Abhängigkeit der Schnittstellenklassen beider Anwendungsfälle. Die umgekehrte, den Zyklus schließende Verknüpfung erfolgt durch Abhängigkeit 9.2.7 - *LeitungsauftragAendernErfassenAA - SVAuswaehlenAA*.

9.2.6 SeminartypAendernErfassenAA - DozentenvereinbarungAuswaehlenAA

9.2.6.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	1	1,6	2,3	0,0	0,0	0,0	0,0	1	3	create and access

Abbildung 47: Abhängigkeitsmetriken SeminartypAendernErfassenAA - DozentenvereinbarungAuswaehlenAA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	8	1	0	0	1	7	0	1	6	0	3	3	4	3	4	4	110	84	81	1,8	2,1
B	8	4	3	1	0	4	0	0	4	2	2	0	4	0	4	4	106	80	77	1,7	2,0

Abbildung 48: Komponentenmetriken SeminartypAendernErfassenAA (A) und DozentenvereinbarungAuswaehlenAA (B)

Erläuterungen zu den Abhängigkeitsmetriken:

Diese Abhängigkeit wiederum ist kein Teil eines Zyklus (dep_comp). Sie verläuft jedoch ebenfalls über Paketgrenzen hinweg (is_intpk).

Der rACD-Wert liegt etwa in Höhe des rACDh-Wertes. Das Entfernen der Abhängigkeit hat keine Auswirkungen auf Zyklen..

Die Abhängigkeit wird durch Erzeugung eines Objekts ausgelöst, auf das dann mehrfach zugegriffen wird (d_type, sub_hw und subedges).

Erläuterungen zu den Komponentenmetriken:

Beide Klassen besitzen eine mittlere Zahl von ausgehenden Abhängigkeiten. Der rACD_out-Wert von SVAendernErfassenAA liegt etwa beim rACD-Wert der Abhängigkeit.

Die Anzahl der direkten und indirekten Abhängigkeiten (cd_tr) beider Klassen liegt im unteren Bereich der Schnittstellenklassen. Die Anzahl statischer Abhängigkeiten liegt insbesondere in der abhängigen Klasse relativ hoch.

9.2.6.2 Bewertung:

In der Anforderungsspezifikation existieren *exceptional flows* zu den Anwendungsfällen *Seminartyp bearbeiten* (a) und *Seminartyp erfassen* (b):

> (a) Die Dozentenvereinbarungen werden bearbeitet (extension point “Personal“).
> (b) Die Dozentenvereinbarungen werden erfasst (extension point “Personal“).

Die in der Anforderungsspezifikation vorgeschriebene Bearbeitung oder Erfassung einer Dozentenvereinbarung wurde beim Entwurf von *SeminarIS* derart realisiert, dass hierdurch eine Abhängigkeit dieses Anwendungsfalles mit den Anwendungsfällen zur Bearbeitung einer Dozentenvereinbarung entstand (siehe Abschnitt 10.8). Dies führt zur Abhängigkeit der zu den Anwendungsfällen implementierten Schnittstellenklassen und den damit verbundenen Testproblemen.

9.2.7 LeitungsauftragAendernErfassenAA - SVAuswaehlenAA

9.2.7.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	1	1	1,6	2,5	13,6	2,6	11,3	20,0	1	2	create and access

Abbildung 49: Abhängigkeitsmetriken LeitungsauftragAendernErfassenAA - SVAuswaehlenAA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	22	10	2	2	6	12	0	1	11	0	6	5	11	5	11	7	184	166	153	2,2	2,9
B	13	1	0	1	0	12	0	1	11	0	7	4	3	2	10	10	184	166	153	4,3	4,9

Abbildung 50: Komponentenmetriken LeitungsauftragAendernErfassenAA (A) und SVAuswaehlenAA (B)

Auffälligkeiten bei den berechneten Werten:

Auffallend ist die sehr hohe Zahl von ausgehenden Abhängigkeiten (cd) bei der Klasse `LeitungsauftragAendernErfassenAA`. Auch die hohen rACD-Werte der Klasse `SVAuswaehlenAA` sind bemerkenswert. Die extrem hohe Anzahl der direkten und indirekten Abhängigkeiten (cd_tr) beider Klassen resultiert unter anderem aus der Beteiligung an verschiedenen Zyklen. Die Anzahl statischer Abhängigkeiten liegt in beiden Klassen sehr hoch (cd_h).

9.2.7.2 Bewertung

In der Anforderungsspezifikation heißt es im *main flow* zum Anwendungsfall *Leitungsauftrag erfassen*:

> Der Seminarsachbearbeiter wählt eine Person und eine **Seminarveranstaltung** aus und erfasst die Angaben zum Leitungsauftrag.

Die in der Anforderungsspezifikation vorgeschriebene Auswahl einer Seminarveranstaltung wurde beim Entwurf von *SeminarIS* derart realisiert, dass hierdurch eine Abhängigkeit dieses Anwendungsfalles mit den Anwendungsfällen zur Bearbeitung einer Seminarveranstaltung entstand (siehe Abschnitt 10.8). Dies führt zur Abhängigkeit der zu den Anwendungsfällen implementierten Schnittstellenklassen und den damit verbundenen Testproblemen.

Da in diesem Fall darüber hinaus auch eine Abhängigkeit vom Anwendungsfall *Seminarveranstaltung erfassen* zum Anwendungsfall *Leitungsauftrag erfassen* besteht, führt dies sogar zu einer zyklischen Abhängigkeit der Schnittstellenklassen beider Anwendungsfälle. Die umgekehrte, den Zyklus schließende Verknüpfung erfolgt durch Abhängigkeit 9.2.5 SVAendernErfassenAA - LeitungsauftragAuswaehlenAA.

9.2.8 DVKostenBerechnenK - DozentenvereinbarungLoeschenK

9.2.8.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	1	0	1,6	1,9	2,5	2,6	0,9	0,0	1	1	static access

Abbildung 51: Abhängigkeitsmetriken DVKostenBerechnenK - DozentenvereinbarungLoeschenK

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	1	1	0	0	5	0	0	5	0	2	3	2	1	4	4	89	65	62	8,8	7,8
B	18	8	5	0	3	10	0	0	10	0	1	9	4	1	14	9	89	65	62	1,8	1,2

Abbildung 52: Komponentenmetriken DVKostenBerechnenK (A) und DozentenvereinbarungLoeschenK (B)

Auffälligkeiten bei den berechneten Werten:

Die Abhängigkeit beruht nur auf einem, allerdings fest verdrahteten Zugriff (sub_hw, subedges). Die rACD-Werte der Klasse `DVKostenBerechnenK` liegen massiv höher als der rACD-Wert der Abhängigkeit. Die Anzahl der von `DozentenvereinbarungLoeschenK` ausgehenden Abhängigkeiten liegt bei einem sehr hohen Wert (cd).

9.2.8.2 Bewertung

In der Anforderungsspezifikation war das Protokollieren von Löschvorgängen gefordert. Somit existiert in jeder Kontrollklasse, die für das Löschen eines Datensatzes verantwortlich ist, entsprechende Protokollfunktionalität. Diese Funktionalität der Klasse `DozentenvereinbarungLoeschenK` wurde hier in fehlerhafter Weise zur Speicherung von Debug- und Fehlermeldungen zweckentfremdet.

Wie bereits bei Abhängigkeit 9.2.1 *Dozentenvereinbarung - DVKostenBerechnenK* erläutert, findet die Klasse `DVKostenBerechnenK` in der Anwendung keine aktive Verwendung. Die Klasse, und damit auch die vorliegende Abhängigkeit, kann ohne Einschränkung der Funktionalität aus dem Projekt entfernt werden.

9.2.9 SeminarbelegungAA - SVAuswaehlenAA

9.2.9.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	1	1,6	2,4	0,0	0,0	0,0	0,0	2	3	create and access

Abbildung 53: Abhängigkeitsmetriken SeminarbelegungAA - SVAuswaehlenAA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	21	8	0	3	5	13	0	1	12	1	8	3	11	6	10	7	190	171	159	1,9	2,7
B	13	1	0	1	0	12	0	1	11	0	7	4	3	2	10	10	184	166	153	4,3	4,9

Abbildung 54: Komponentenmetriken SeminarbelegungAA (A) und SVAuswaehlenAA (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `SVAuswaehlenAA` sind die fünfthöchsten aller Klassen. Die Anzahl der von `SeminarbelegungAA` ausgehenden Abhängigkeiten liegt bei einem sehr hohen Wert (cd). Durch direkte und indirekte Beteiligung an verschiedenen Zyklen handelt es sich ferner bei den Anzahlen der Klassen, von denen beide Klassen abhängig sind (cd_tr), um mit die höchsten Werte des Systems.

9.2.9.2 Bewertung

In der Anforderungsspezifikation heißt es im *main flow* zum Anwendungsfall *Teilnehmer anmelden*:

> Eine Person meldet sich telefonisch oder schriftlich für eine öffentliche Seminarveranstaltung an. Der Kundensachbearbeiter prüft, ... ob die gewünschte Seminarveranstaltung existiert ... Die Seminargebühr wird vereinbart und eine entsprechende Belegung erzeugt.

Die in der Anforderungsspezifikation vorgeschriebene Auswahl einer Seminarveranstaltung wurde beim Entwurf von *SeminarIS* derart realisiert, dass hierdurch eine Abhängigkeit dieses Anwendungsfalles mit den Anwendungsfällen zur Bearbeitung einer Seminarveranstaltung entstand (siehe Abschnitt 10.8). Dies führt zur Abhängigkeit der zu den Anwendungsfällen implementierten Schnittstellenklassen und den damit verbundenen Testproblemen.

9.2.10 SeminarisH - SachbearbeiterA

9.2.10.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	1	0	1,5	2,4	1,2	2,6	0,9	0,0	1	3	static access

Abbildung 55: Abhängigkeitsmetriken SeminarisH - SachbearbeiterA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	2	0	0	0	0	2	0	0	2	0	1	1	2	2	0	0	89	65	62	2,4	1,8
B	9	4	1	1	2	5	0	0	5	0	2	3	2	1	7	4	89	65	62	1,9	1,0

Abbildung 56: Komponentenmetriken SeminarisH (A) und SachbearbeiterA (B)

Auffälligkeiten bei den berechneten Werten:

Die Klasse `SeminarisH` besitzt als Hauptschnittstellenklasse nur eine geringe Zahl von Abhängigkeiten zu anderen Klassen (`cd`). Beide Abhängigkeiten sind allerdings statischer Natur (`cd_h`).

9.2.10.2 Bewertung

In der Hauptschnittstellenklasse `SeminarisH` existiert die statische Variable `MainWindow`, die bei der Erzeugung von `SeminarisH` mit der einzigen Instanz des Hauptanwendungsfensters `SachbearbeiterA` initialisiert wird. Auch hier wird in problematischer Weise die Absicht verfolgt, einen globalen Zugriff auf diese Komponente zur Verfügung zu stellen (siehe Diskussion in Abschnitt 10.6).

Verschärft wird dieses Problem erneut dadurch, dass auch aus der tiefer liegenden Kontrollschicht auf diese statische Variable zugegriffen wird und damit die Drei-Schicht-Architektur verletzt wird. Lediglich der in der Klasse SachbearbeiterA realisierte *Reflection*-Aufruf der initialen Schnittstellenklassen verhindert die zyklische Abhängigkeit nahezu aller Systemkomponenten (siehe Abschnitt 10.3).

9.2.11 ErweitertPersistent - SeminarisDatenbank

9.2.11.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	1	1,4	4,4	3,7	7,7	0,0	-40,0	1	1	static access

Abbildung 57: Abhängigkeitsmetriken ErweitertPersistent - SeminarisDatenbank

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	2	1	1	0	0	1	0	0	1	0	0	1	1	0	1	1	89	65	62	2,0	1,4
B	12	5	4	0	1	7	0	0	7	1	0	6	2	2	10	5	89	65	62	17,7	16,8

Abbildung 58: Komponentenmetriken ErweitertPersistent (A) und SeminarisDatenbank (B)

Auffälligkeiten bei den berechneten Werten:

Die Wert der Metriken `rACD` und `rACDh` ist stark beeinflusst vom Zugriff der Klasse `SeminarisDatenbank` auf die Datenhaltungsschicht (siehe Abhängigkeit 9.2.2 und Abschnitt

10.5). Sobald die Abhängigkeit der Datenbank von der Datenhaltung beseitigt ist, reduzieren sich beide Metrikwerte deutlich (siehe Abschnitt 11.4.3).

Die Entfernung der Abhängigkeit würde zu einer Zersplittung des Datenhaltungszyklus in drei Teile führen (rNDC).

9.2.11.2 Bewertung

Sämtliche Klassen, deren Instanzen persistent in der Datenbank gehalten werden sollen (Domänenklassen sowie Paar- und Tripelklassen des Transformationenpakets), erweitern die abstrakte Klasse `ErweitertPersistent`, die über die Implementierung des Interfaces `IErweitertPersistent` auch das Interface `IPersistent` des Persistenz-Rahmenwerks implementiert. Zur Erzeugung persistenter Objekte existiert in diesen Klassen die Methode `erzeuge()`, die durch Aufruf der Methode `erzeugeObjekt( IPersistent templateKlasse)` der Datenbankklasse des Persistenz-Rahmenwerks eine persistente Instanz der jeweiligen Klasse zurückgibt.

Da folglich in sämtlichen von `ErweitertPersistent` abgeleiteten Klassen der Zugriff auf die Datenbank benötigt wird, wurde in `ErweitertPersistent` die Methode `getDb()` implementiert, die fest verdrahtet die *Singleton*-Instanz der `SeminarisDatenbank` liefert:

```
protected static Datenbank getDb() {
    return SeminarisDatenbank.getEinzigeInstanz();
}
```

Hier handelt es sich um eine weitere Form der Realisierung eines globalen Zugriffs auf eine Basiskomponente des Systems (vergleichbar dem Zugriff auf die Datenbank aus der Kontrollschicht in Abhängigkeit 9.2.2). Durch den fest verdrahteten Zugriff kann ein von der Datenbank isolierter Test somit nur durch eine Änderung der Implementierung in der Klasse `ErweitertPersistent` oder durch den Austausch der Klasse `SeminarisDatenbank` stattfinden (siehe Abschnitt 10.6).

9.2.12 SVKurzS - Seminarveranstaltung

9.2.12.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	1	1,1	-	0,0	0,0	0,0	0,0	0	8	dep. to abstract class

Abbildung 59: Abhängigkeitsmetriken SVKurzS - Seminarveranstaltung

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	3	3	1	2	0	0	0	0	0	0	0	0	3	0	0	0	91	0	0	0,5	1,1
B	12	5	5	0	0	7	1	0	6	1	1	4	7	3	5	4	89	65	62	2,1	1,5

Abbildung 60: Komponentenmetriken SVKurzS (A) und Seminarveranstaltung (B)

Auffälligkeiten bei den berechneten Werten:

Beim Blick auf die Abhängigkeitsmetriken fällt auf, dass es sich hier um eine nicht-statische Abhängigkeit zu einer abstrakten Klasse handelt (sub_hw, d_type). Bemerkenswert ist ferner, dass die abhängige Klasse nur eine sehr geringe Zahl von Abhängigkeiten besitzt (cd), und sich darunter keine einzige statische Abhängigkeit befindet (cd_h).

9.2.12.2 Bewertung

Bei der Klasse `SVKurzS` handelt es sich um die Kurzform einer Sichtschnittstellenklasse, die in fremden Anwendungsfällen (*Leitungsauftrag erfassen*, *Teilnehmer anmelden*) zur Zuordnung einer Seminarveranstaltung genutzt wird. Die vorliegende Abhängigkeit verläuft zu der von ihr angezeigten Domänenklasse `Seminarveranstaltung`.

Dieser Abhängigkeit liegt eine durchaus übliche und auch empfohlene Vorgehensweise zugrunde [SWEII99]. Die Übergabe der zu präsentierenden Seminarveranstaltung erfolgt als Parameter an die Klasse `SVKurzS`. Da es sich beim Typ des Übergabeparameters um eine abstrakte Klasse handelt, scheint auf Anhieb durch eine Änderung der Implementierung keine Verbesserung der Testbarkeit erreichbar.

9.2.13 MeldungsAnzeigeA - ExceptionNachrichtA

9.2.13.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	0	1,0	0,8	0,0	0,0	0,0	0,0	1	3	create and access

Abbildung 61: Abhängigkeitsmetriken MeldungsAnzeigeA - ExceptionNachrichtA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	5	2	0	1	1	3	0	0	3	0	2	1	1	0	4	3	89	65	62	2,9	2,3
B	2	2	0	1	1	0	0	0	0	0	0	0	1	0	1	0	2	0	0	1,0	0,0

Abbildung 62: Komponentenmetriken MeldungsAnzeigeA (A) und ExceptionNachrichtA (B)

Auffälligkeiten bei den berechneten Werten:

Die Klasse `ExceptionNachrichtA` gehört zu den Klassen mit den wenigsten direkten und indirekten Abhängigkeiten im System (cd_tr).

9.2.13.2 Bewertung

Bei den Klassen `MeldungsAnzeigeA` und `ExceptionNachrichtenA` handelt es sich um Schnittstellenklassen zur Ausgabe von Hinweisen und Fehlermeldungen. Die Klasse `ExceptionNachrichtenA` wird ausschließlich von der Klasse `MeldungsAnzeigeA` genutzt, um die Ausgabe der Fehlermeldung in abgewandelter Form vorzunehmen.

Für die Verbesserung der Testbarkeit jener Methode der Klasse `MeldungsAnzeigeA`, in der die Abhängigkeit ausgelöst wird, würde die Entfernung keinen wesentlichen Beitrag leisten.

9.2.14 TeilnahmeFirmenSVRelation - TeilnahmeFirmenSVPaar

9.2.14.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 63: Abhängigkeitsmetriken TeilnahmeFirmenSVRelation - TeilnahmeFirmenSVPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	3	3	0	0	3	1	0	2	1	1	0	3	1	3	2	89	65	62	2,9	2,3
B	5	3	3	0	0	2	1	0	1	0	0	1	4	1	1	1	89	65	62	1,0	0,4

Abbildung 64: Komponentenmetriken TeilnahmeFirmenSVRelation (A) und TeilnahmeFirmenSVPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `TeilnahmeFirmenSVRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.14.2 Bewertung

Die Klassen `TeilnahmeFirmenSVRelation` und `TeilnahmeFirmenSVPaar` resultieren aus der Transformation der Assoziation zwischen einem Firmenseminar und einer daran teilnehmenden Person. Die Transformationstechnik ist ein übliches Muster und wurde nach den Vorgaben durchgeführt **[**SWEII99**]**.

Während üblicherweise allerdings der Zugriff auf die Relationenklasse aus der Kontrollschicht direkt erfolgt, fungierten in *SeminarIS* die Domänenklassen als Fassade für den Zugriff auf die Assoziation. Dies führte zu einer zyklischen Vernetzung der nahezu kompletten Datenhaltungsschicht (siehe Abschnitt 10.7).

Die Abhängigkeiten von den Relationenklassen zu den Paarklassen sind gute Indikatoren für diese Problematik, da der Zugriff auf die Paarklasse exklusiv durch die Relationenklasse erfolgt. Dadurch entstehen in der Datenhaltungsschicht zwischen den jeweils zusammengehörenden Relation- und Paarklassen die zu Abhängigkeit 9.2.2 beschriebenen *Hub*-Abhängigkeiten.

9.2.15 SVSeminartypRelation - SVSeminartypPaar

9.2.15.1 Ergebnisse der Metrikberechnung

Man erhält folgende Abhängigkeits- und Komponentenmetriken:

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 65: Abhängigkeitsmetriken SVSeminartypRelation - SVSeminartypPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	5	3	3	0	0	2	1	0	1	0	1	0	3	1	2	1	89	65	62	2,9	2,3
B	5	3	3	0	0	2	1	0	1	0	0	1	4	1	1	1	89	65	62	1,0	0,4

Abbildung 66: Komponentenmetriken SVSeminartypRelation (A) und SVSeminartypPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `SVSeminartypRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.15.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Seminarveranstaltung und einem Seminartyp. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen.

9.2.16 SVAnsprechpartnerRelation - SVAnsprechpartnerPaar

9.2.16.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 67: Abhängigkeitsmetriken SVAnsprechpartnerRelation - SVAnsprechpartnerPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	3	3	0	0	3	1	0	2	1	1	0	3	1	3	2	89	65	62	2,9	2,3
B	5	3	3	0	0	2	1	0	1	0	0	1	4	1	1	1	89	65	62	1,0	0,4

Abbildung 68: Komponentenmetriken SVAnsprechpartnerRelation (A) und SVAnsprechpartnerPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `SVAnsprechpartnerRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.16.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Seminarveranstaltung und einer als Ansprechpartner fungierenden Person. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen.

9.2.17 RechnungsempfaengerRelation - RechnungsempfaengerPaar

9.2.17.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	2,6	0,9	0,0	2	2	create and access

Abbildung 69: Abhängigkeitsmetriken RechnungsempfaengerRelation - RechnungsempfaengerPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	9	4	3	0	1	5	1	0	4	1	1	2	4	2	5	3	89	65	62	2,9	2,3
B	4	3	3	0	0	1	0	1	0	0	0	0	4	1	0	0	89	65	62	1,0	0,4

Abbildung 70: Komponentenmetriken RechnungsempfaengerRelation (A) und RechnungsempfaengerPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `RechnungsempfaengerRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.17.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Belegung und einem als Rechnungsempfänger für die Belegung eingetragenen Geschäftspartner. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen.

9.2.18 SeminarisK - LogProtokollDateiA

9.2.18.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	1	1	1,0	1,2	1,2	2,6	0,9	0,0	1	2	create and access

Abbildung 71: Abhängigkeitsmetriken SeminarisK - LogProtokollDateiA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	0	0	0	0	6	0	0	6	0	2	4	6	6	0	0	89	65	62	8,8	8,2
B	4	2	0	0	2	2	1	0	1	0	0	1	1	1	3	1	89	65	62	1,0	0,4

Abbildung 72: Komponentenmetriken SeminarisK (A) und LogProtokollDateiA (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `SeminarisK` liegen massiv höher als der rACD-Wert der Abhängigkeit. Sämtliche von `SeminarisK` ausgehenden Abhängigkeiten sind im Sourcecode fest verdrahtet.

9.2.18.2 Bewertung

Bei der Klasse `LogProtokollDateiA` handelt es sich um die Implementierung des Anwendungsprotokolls von *SeminarIS*. Im Anwendungsprotokoll wird nach Vorgabe der Anforderungsspezifikation das Löschen von Entitäten vermerkt. Das Erzeugen der Klasse `LogProtokollDateiA` erfolgt in der Hauptkontrollklasse `SeminarisK`.

Bei ihrem Erzeugen registriert sich die Klasse `LogProtokollDateiA` beim Protokollmanager von *SeminarIS*., der für die Verteilung der Protokolleinträge an die bei ihm angemeldeten Instanzen verantwortlich ist. Bei ihrer Registrierung verwendet die Klasse `LogProtokollDateiA` für den Zugriff auf die Klasse `ProtokollManager` die globale Variable `Protocol` der Klasse `SeminarisK`.

Dies führt zu der in Abschnitt 10.4 beschriebenen Abhängigkeit einer Basiskomponente von der Kontrollschicht, die in diesem Fall auch noch einen Zyklus erzeugt. Durch die Bündelung globaler Variablen in der Klasse `SeminarisK` wird dadurch insbesondere auch die indirekte Abhängigkeit der Protokolldatei von der Datenbank und in der Folge von der Datenhaltungsschicht bewirkt (siehe Abhängigkeit 9.2.2 *SeminarisK - SeminarisDatenbank*). Ursache für die Bündelung der globalen Variablen ist der Versuch, globalen Zugriff auf die Basiskomponenten zur Verfügung zu stellen (siehe Abschnitt 10.6).

9.2.19 SeminarisK - DebugProtokollDateiA

9.2.19.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	1	1	1,0	1,2	1,2	2,6	0,9	0,0	1	2	create and access

Abbildung 73: Abhängigkeitsmetriken SeminarisK - DebugProtokollDateiA

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	0	0	0	0	6	0	0	6	0	2	4	6	6	0	0	89	65	62	8,8	8,2
B	4	2	0	0	2	2	1	0	1	0	0	1	1	1	3	1	89	65	62	1,0	0,4

Abbildung 74: Komponentenmetriken SeminarisK (A) und DebugProtokollDateiA (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `SeminarisK` liegen massiv höher als der rACD-Wert der Abhängigkeit. Sämtliche von `SeminarisK` ausgehenden Abhängigkeiten sind im Sourcecode fest verdrahtet.

9.2.19.2 Bewertung

Bei der Klasse `DebugProtokollDateiA` handelt es sich um die Implementierung des internen Fehlerprotokolls von *SeminarIS*. Zur Verwendung des Fehlerprotokolls wurden allerdings keine Richtlinien festgelegt, so dass die Verwendung nur sporadisch von einigen Entwicklern erfolgte.

Die Implementierung der Klasse `DebugProtokollDateiA` erfolgte völlig analog zur Klasse `LogProtokollDateiA`. Bezüglich der Einordnung der Abhängigkeit siehe demzufolge vorstehenden Abschnitt 9.2.18.

9.2.20 AnstellungRelation - AnstellungPaar

9.2.20.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 75: Abhängigkeitsmetriken AnstellungRelation - AnstellungPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	3	3	0	0	3	1	0	2	1	1	0	3	1	3	2	89	65	62	2,9	2,3
B	4	3	3	0	0	1	0	1	0	0	0	0	4	1	0	0	89	65	62	1,0	0,4

Abbildung 76: Komponentenmetriken AnstellungRelation (A) und AnstellungPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `AnstellungRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.20.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Firma und einer bei ihr angestellten Person. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen.

9.2.21 BelegungRelation - BelegungTripel

9.2.21.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 77: Abhängigkeitsmetriken BelegungRelation - BelegungTripel

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	13	7	6	0	1	6	1	0	5	1	2	2	5	2	8	4	89	65	62	2,9	2,3
B	5	4	4	0	0	1	0	1	0	0	0	0	5	1	0	0	89	65	62	1,0	0,4

Abbildung 78: Komponentenmetriken BelegungRelation (A) und BelegungTripel (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `BelegungRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.21.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer öffentlichen Seminarveranstaltung, einer daran teilnehmenden Person und der Assoziationsklasse `Belegung`. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen. Die Abhängigkeiten zwischen Relationen- und Tripelklassen sind völlig analog zu sehen.

9.2.22 BuchtRelation - BuchtPaar

9.2.22.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 79: Abhängigkeitsmetriken BuchtRelation - BuchtPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	3	3	0	0	3	1	0	2	1	1	0	3	1	3	2	89	65	62	2,9	2,3
B	5	3	3	0	0	2	1	0	1	0	0	1	4	1	1	1	89	65	62	1,0	0,4

Abbildung 80: Komponentenmetriken BuchtRelation (A) und BuchtPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `BuchtRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.22.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Firma und einer von ihr gebuchten Seminarveranstaltung. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen.

9.2.23 FirmenAnsprechpartnerRelation - FirmenAnsprechpartnerPaar

9.2.23.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	2	2	create and access

Abbildung 81: Abhängigkeitsmetriken FirmenAnsprechpartnerRelation - FirmenAnsprechpartnerPaar

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	6	3	3	0	0	3	1	0	2	1	1	0	3	1	3	2	89	65	62	2,9	2,3
B	4	3	3	0	0	1	0	1	0	0	0	0	4	1	0	0	89	65	62	1,0	0,4

Abbildung 82: Komponentenmetriken FirmenAnsprechpartnerRelation (A) und FirmenAnsprechpartnerPaar (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `FirmenAnsprechpartnerRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.23.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Firma und einer bei ihr als Ansprechpartner fungierenden Person. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen.

9.2.24 LeitungsauftragRelation - LeitungsauftragTripel

9.2.24.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	0,0	0,0	0,9	0,0	3	3	create and access

Abbildung 83: Abhängigkeitsmetriken LeitungsauftragRelation - LeitungsauftragTripel

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	8	4	4	0	0	4	1	0	3	1	2	0	4	1	4	3	89	65	62	2,9	2,3
B	6	4	4	0	0	2	1	0	1	0	0	1	5	1	1	1	89	65	62	1,0	0,4

Abbildung 84: Komponentenmetriken LeitungsauftragRelation (A) und LeitungsauftragTripel (B)

Auffälligkeiten bei den berechneten Werten:

Die rACD-Werte der Klasse `LeitungsauftragRelation` liegen deutlich über dem rACD-Wert der Abhängigkeit.

9.2.24.2 Bewertung

Die beiden beteiligten Klassen resultieren aus der transformierten Assoziation zwischen einer Person, einer von ihr geleiteten Seminarveranstaltung und der Assoziationsklasse `Leitungsauftrag`. Die Bewertung der Abhängigkeiten zwischen Relationen- und Paarklassen wurde bereits bei Abhängigkeit 9.2.14 vorgenommen. Die Abhängigkeiten zwischen Relationen- und Tripelklassen sind völlig analog zu sehen.

9.2.25 DozentenvereinbarungRelation - DozentenvereinbarungAnfragen

9.2.25.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	1,2	1,2	0,0	0,9	0,0	6	7	static access

Abbildung 85: Abhängigkeitsmetriken DozentenvereinbarungRelation - DozentenvereinbarungAnfragen

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	7	5	4	0	1	2	0	0	2	0	1	1	5	1	2	1	89	65	62	1,9	1,3
B	5	3	3	0	0	2	0	0	2	0	0	2	2	0	3	2	89	65	62	1,0	0,4

Abbildung 86: Komponentenmetriken DozentenvereinbarungRelation (A) und DozentenvereinbarungAnfragen (B)

Auffälligkeiten bei den berechneten Werten:

Die Anzahl der Statements, die Auslöser für die vorliegende Abhängigkeit sind, ist relativ hoch (subedges).

9.2.25.2 Bewertung

Abweichend von der Standardvorgabe bei der Realisierung von Relationenklassen wurde bei der Transformation der Assoziation Dozentenvereinbarung eine zusätzliche Klasse `DozentenvereinbarungAnfragen` eingeführt. In diese Klasse wurden alle Abfragen auf die Datenbank ausgelagert. Bei allen anderen Assoziationen sind diese Datenbankabfragen innerhalb der Relationenklasse realisiert.

Der Zugriff auf die Klasse `DozentenvereinbarungAnfragen` erfolgt exklusiv durch die Klasse `DozentenvereinbarungRelation`. Zur Ansteuerung der Datenbank greift die Klasse `DozentenvereinbarungAnfragen` aus der Datenhaltungsschicht auf die höher liegende Kontrollschicht zu. Dies führt zur Verletzung der Hierarchie in der Drei-Schicht-Architektur (siehe Abschnitt 10.3 *Verletzung der Drei-Schicht-Architektur*). Verursacht wird dies im Bemühen, auf eine Basiskomponente zuzugreifen (siehe Abschnitt 10.6 *Zugriff auf globale Basiskomponenten*).

Insgesamt führt dies wie bei Abhängigkeit 9.2.14 *TeilnahmeFirmenSVRelation - TeilnahmeFirmenSVPaar* zur *Hub*-Eigenschaft dieser Abhängigkeit innerhalb der zyklischen Datenhaltungsschicht, wodurch die erhöhte rACD-Metrik der Abhängigkeit zustande kommt und die Anzeige als testkritische Abhängigkeit erfolgen kann.

9.2.26 DozentenvereinbarungTripel - DVSchluessel

9.2.26.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	0	1,0	0,8	1,2	0,0	0,9	0,0	5	7	create and access

Abbildung 87: Abhängigkeitsmetriken DozentenvereinbarungTripel - DVSchluessel

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	12	5	4	0	1	7	1	0	6	1	1	4	9	4	3	3	89	65	62	1,9	1,3
B	3	3	3	0	0	0	0	0	0	0	0	0	2	0	1	0	89	0	0	1,0	0,4

Abbildung 88: Komponentenmetriken DozentenvereinbarungTripel (A) und DVSchluessel (B)

Auffälligkeiten bei den berechneten Werten:

Die Anzahl der Statements, die Auslöser für die vorliegende Abhängigkeit sind, ist relativ hoch (subedges). Die Klasse `DVSchluessel` ist frei von statischen Abhängigkeiten (cd_h, cd_h_tr).

9.2.26.2 Bewertung

Die Definition der Klassen `DozentenvereinbarungTripel` und `DVSchluessel` erfolgt in derselben Sourcecodedatei. Die Klasse `DVSchluessel` stellt verschiedene Operationen zum Umgang mit den innerhalb der Anwendung verwendeten Schlüsseln persistenter Instanzen zur Verfügung. Der Zugriff auf die Klasse `DVSchluessel` erfolgt exklusiv aus der Klasse `DozentenvereinbarungTripel`, was wiederum zur *Hub*-Eigenschaft der Abhängigkeit führt (vergleiche Abhängigkeit 9.2.14 *TeilnahmeFirmenSVRelation - TeilnahmeFirmenSVPaar*).

Bei dieser Klasse handelt es sich ihrem Verwendungszweck nach eigentlich um eine innere Klasse der Tripelklasse. Demzufolge sollte sie auch unabhängig von der Refaktorisierung der Datenhaltung so modelliert werden. Dies hätte allerdings nur Einfluss auf die Metrikwerte und nicht auf die Testbarkeit des Programms.

9.2.27 SeminarveranstaltungOrdner - OeffentlicheSeminarveranstaltung

9.2.27.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	1	1	0,8	1,1	1,2	2,6	0,9	0,0	1	1	static access

Abbildung 89: Abhängigkeitsmetriken SeminarveranstaltungOrdner - OeffentlicheSeminarveranstaltung

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	5	0	0	0	0	5	1	0	4	0	0	4	1	1	4	4	89	65	62	1,8	1,2
B	13	7	6	1	0	6	0	1	5	0	1	4	6	2	7	4	89	65	62	1,0	0,4

Abbildung 90: Komponentenmetriken SeminarveranstaltungOrdner (A) und OeffentlicheSeminarveranstaltung (B)

Auffälligkeiten bei den berechneten Werten:

Die Abhängigkeit beruht nur auf einem einzigen, allerdings statischen Zugriff (sub_hw, subedges, d_type). Alle von der Klasse `SeminarveranstaltungOrdner` ausgehenden Abhängigkeiten sind im Sourcecode fest verdrahtet (cd_h).

9.2.27.2 Bewertung

Die Ordnerklassen in *SeminarIS* sind bezüglich ihrer Funktionalität auf die Realisierung der Datenbankabfragen reduziert. Die Verwaltung der persistenten Instanzen und der Zugriff auf diese Instanzen erfolgt über die Domänenklassen. Die in diesem Zusammenhang realisierte Fassadenfunktionalität der Domänenklassen wird in Abschnitt 10.7 ausführlicher beschrieben.

Diese Abhängigkeit veranschaulicht noch einmal die in diesem Zusammenhang auftretenden Probleme. Die Domänenklasse `OeffentlicheSeminarveranstaltung` besitzt diverse Methoden, die im Sinne einer Fassade den Methodenaufruf an die Ordnerklasse durchreicht. Nachfolgend ein Beispiel für diese Art der Methode:

```
public static Iterator getAlle() throws DatenbankAusnahme {
   return SeminarveranstaltungOrdner.getEinzigeInstanz().getOeSVs();
}
```

Um die Ergebnismenge aus der Datenbank zu lesen, erfolgt in der aufgerufenen Methode der Ordnerklasse unter Verwendung des Persistenz-Rahmenwerks eine entsprechende Abfrage auf die Datenbank, für die als Muster die Klasse `OeffentlicheSeminarveranstaltung` verwendet werden muss:

```
public Iterator getOeSVs() throws DatenbankAusnahme {
   try {
      Abfrage abfrage = new Abfrage(OeffentlicheSeminarveranstaltung.class);
      ...
      Iterator iter = getDb().abfrageAusfuehren(abfrage);
      ...
```

Man sieht, wie auf diese Weise eine zyklische Abhängigkeit zwischen Ordner- und Domänenklasse entsteht, die bei Beibehaltung der Fassadenfunktion der Domänenklasse kaum vermieden werden kann.

Aus Sicht der Testbarkeit führt dies zu einer unnötigen Verschmelzung der Domänen- mit den Ordnerklassen, die einen isolierten Test der Domänenklassen unmöglich macht.

9.2.28 DozentenvereinbarungLoeschenK - IDozentenvereinbarungK

9.2.28.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	0	0,8	1,0	0,0	0,0	0,0	0,0	1	1	static access

Abbildung 91: Abhängigkeitsmetriken DozentenvereinbarungLoeschenK - IDozentenvereinbarungK

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	18	8	5	0	3	10	0	0	10	0	1	9	4	1	14	9	89	65	62	1,8	1,2
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,6	0,0

Abbildung 92: Komponentenmetriken DozentenvereinbarungLoeschenK (A) und IDozentenvereinbarungK (B)

Auffälligkeiten bei den berechneten Werten:

Die Abhängigkeit beruht nur auf einem einzigen, allerdings statischen Zugriff (sub_hw, subedges, d_type). Die Klasse, zu der die Abhängigkeit führt, besitzt keine weiteren Abhängigkeiten (cd).

9.2.28.2 Bewertung

Der Entwurf und die Implementierung der Schnittstellen- und Kontrollklassen, die sich aus den Anwendungsfällen zum Bearbeiten, Erfassen und Löschen von Dozentenvereinbarung ergaben, erfolgte abweichend von den in der Entwurfsphase aufgestellten Design-Richtlinien der Entwicklergruppe. So existiert beispielsweise in der Kontrollschicht ein Interface `IDozentenvereinbarungK`, das von den Kontrollklassen zum Erfassen und Ändern einer Dozentenvereinbarung implementiert wird.

Ferner wird in diesem Interface eine Konstante zur Bestimmung der Protokollpuffergröße für Fehlermeldungen und Einträge in das Anwendungsprotokoll definiert. Durch den Zugriff der Klasse `DozentenvereinbarungLoeschenK` auf diese Konstante entsteht die vorliegende Abhängigkeit.

Die von den Richtlinien der Gruppe abweichende Umsetzung der Anwendungsfälle zur Dozentenvereinbarung wie auch der zugehörigen Teile der Datenhaltung widerspricht allen Prinzipien des objektorientierten Entwurfs. Aus diesem Grund ist es sehr schwierig, alle in diesem Zusammenhang auftretenden Abhängigkeiten auf ihre Notwendigkeit und Sinnhaftigkeit zu prüfen und Alternativvorschläge zu geben.

Die Testkritikalität dieser Abhängigkeit entsteht als Folge der Abhängigkeiten 9.2.1 *Dozentenvereinbarung - DVKostenBerechnenK* und 9.2.8 *DVKostenBerechnenK - DozentenvereinbarungLoeschenK*. Für sich alleine ist sie weitestgehend unbedenklich und auf einen Vorschlag zur Refaktorisierung der Abhängigkeit wird deshalb aus den vorgenannten Gründen verzichtet.

9.2.29 Seminartyp - SeminartypOrdner

9.2.29.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
1	0	1	0,8	1,0	1,2	0,0	0,9	0,0	2	4	static access

Abbildung 93: Abhängigkeitsmetriken Seminartyp - SeminartypOrdner

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	11	5	5	0	0	6	0	1	5	2	0	3	6	2	5	4	89	65	62	2,5	1,9
B	2	0	0	0	0	2	1	0	1	0	0	1	1	1	1	1	89	65	62	1,0	0,4

Abbildung 94: Komponentenmetriken Seminartyp (A) und SeminartypOrdner (B)

9.2.29.2 Bewertung

Auch diese Abhängigkeit entsteht unmittelbar aus der in Abschnitt 10.7 beschriebenen Fassadenfunktionalität der Domänenklassen. Die Klasse `Seminartyp` fungiert als durchreichende Stelle für verschiedene Methodenaufrufe ihrer Ordnerklasse.

Zur weiteren Bewertung dieser Form der Abhängigkeit siehe Abhängigkeit 9.2.27 *SeminarveranstaltungOrdner - OeffentlicheSeminarveranstaltung.*

9.2.30 ParameterEingabeModel - IAnfrageParameter

9.2.30.1 Ergebnisse der Metrikberechnung

dep_comp	is_feedb	is_intpk	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	subedges	d_type
0	0	1	0,8	-	0,0	0,0	0,0	0,0	0	6	dep. to interface

Abbildung 95: Abhängigkeitsmetriken ParameterEingabeModel - IAnfrageParameter

Klasse	cd	cd_t	cd_t_if	cd_t_ac	cd_t_c	cd_h	cd_h_i	cd_h_is	cd_h_s	cd_s_c	cd_s_cu	cd_s_a	cd_ci	cd_ci_h	cd_im	cd_im_h	cd_tr	cd_h_tr	cd_is_tr	rACD_in	rACD_out
A	2	1	1	0	0	1	1	0	0	0	0	0	1	1	1	0	98	1	0	0,4	0,8
B	3	2	1	0	1	1	1	0	0	0	0	0	3	1	0	0	96	1	0	0,9	1,2

Abbildung 96: Komponentenmetriken ParameterEingabeModel (A) und IAnfrageParameter (B)

Auffälligkeiten bei den berechneten Werten:

Bei dieser Abhängigkeit handelt es sich um die idealtypische Abhängigkeit einer (konkreten) Klasse zu einem (abstrakten) Interface (d_type). Die Abhängigkeit ist frei von statischen Zugriffen (sub_hw, subedges).

9.2.30.2 Bewertung

Diese Abhängigkeit ist aus Sicht der Testbarkeit nicht zu beanstanden. Der hohe rACD-Wert für diese Abhängigkeit resultiert aus der Erweiterung des Interfaces `IErweitertPersistent` durch das Interface `IAnfrageParameter`. In der Folge entsteht über die Abhängigkeit von `IErweitertPersistent` zu `ErweitertPersistent` die Abhängigkeit von der Datenbank und Datenhaltung (siehe Abschnitt 10.6 *Zugriff auf globale Basiskomponenten*).

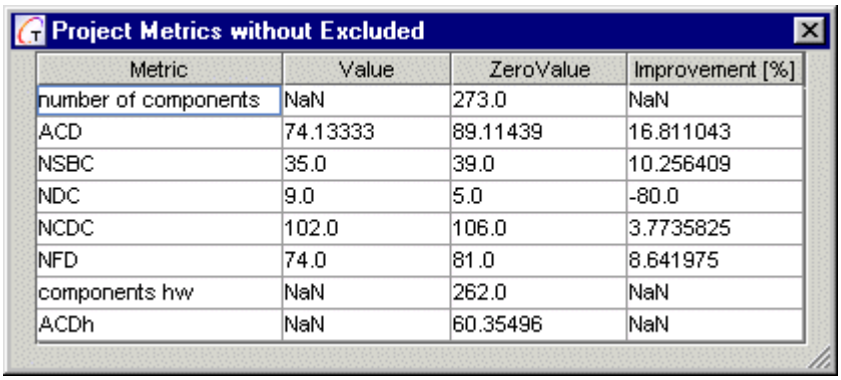
9.3 Adaption der Testbarkeitsmetriken

Die nähere Beschäftigung mit den Ursachen einzelner Abhängigkeiten hat gezeigt, dass die Metriken in manchen Fällen zu granular angelegt sind (siehe Abhängigkeit 9.2.2 SeminarisK - SeminarisDatenbank). Hierdurch kommt es zu „Maskierungseffekten“, wenn beim Vorhandensein paralleler Abhängigkeiten die Testkritikalität dieser Abhängigkeiten nicht erkannt werden kann.

Eine Verbesserung verspricht die für die Zukunft vorgesehene Berechnung der Metriken auf Paketebene. Jedoch können - je nach Art der Realisierung - auch dort Maskierungseffekte auftreten. So verlaufen zwar beispielsweise die Abhängigkeiten aus `SeminarisDatenbank` in die Datenhaltung zwischen denselben Paketen (siehe erneut Abhängigkeit 9.2.2 SeminarisK - SeminarisDatenbank). Die Abhängigkeiten aus der Datenhaltung in die Kontrollschicht verlaufen jedoch ebenfalls parallel zwischen verschiedenen Paketen. Dort kann man mit Effekten rechnen, die denen auf der Klassenebene gleichen.

Ein einfacher Ansatz, um die oben beschriebenen Maskierungseffekte auf Klassenebene zu umgehen, erfolgte durch die Einführung zusätzlicher Komponentenmetriken (rACD_in und rACD_out). Die neue Metrik rACD_out errechnet beispielsweise die Veränderung des ACD-Wertes des Gesamtprojekts nach Entfernung aller von dieser Komponente ausgehenden Abhängigkeiten (die Komponente wird also zur Senke im Abhängigkeitsgraphen).

Beispielsweise erhält man nach Entfernung aller Abhängigkeiten, die als abhängige Komponente `SeminarisDatenbank` besitzen, folgende Projektmetriken:



Metric	Value	ZeroValue	Improvement [%]
number of components	NaN	273.0	NaN
ACD	74.13333	89.11439	16.811043
NSBC	35.0	39.0	10.256409
NDC	9.0	5.0	-80.0
NCDC	102.0	106.0	3.7735825
NFD	74.0	81.0	8.641975
components hw	NaN	262.0	NaN
ACDh	NaN	60.35496	NaN

Abbildung 97: Entfernung der Abhängigkeiten von der Datenbank zur Datenhaltung

Bemerkenswert ist hierbei, dass nur eine der 12 ausgehenden Abhängigkeiten von `SeminarisDatenbank` überhaupt einen von 0 verschiedenen rACD-Wert besitzt, wobei selbst dieser für sich alleine völlig unkritisch ist (0,6). Und doch ergibt sich für den aggregierten rACD_out-Wert der Komponente ein extrem hoher Wert von 16,8. Diese Metrik kann zwar keine treffsichere Angabe zu den tatsächlich ACD-kritischen Abhängigkeiten machen, bietet aber einen guten Anhaltspunkt für das Aufspüren verdeckter Abhängigkeiten.

10 Entwurfsprobleme und Testbarkeit

10.1 Einleitung

Die Bewertung testkritischer Abhängigkeiten im letzten Kapitel und erste Versuche zur Erstellung von Testfällen für *SeminarIS* zeigten sehr schnell, dass bei der Durchführung von Klassentests (*Unit Tests)* die Idealvorstellung des isolierten Tests einer Klasse nicht einmal ansatzweise realisiert werden konnte. Zur Ergründung der Ursache muss die tatsächliche Vorgehensweise bei der Erstellung des Projekts noch einmal rückschauend beleuchtet werden:

- Die Verteilung der Teilaufgaben innerhalb der achtköpfigen Gruppe erfolgte in vertikaler Struktur. Jedes der Projektmitglieder hatte (in der Regel an den Anwendungsfällen orientiert) in jeder der drei Architekturschichten Teilkomponenten zu realisieren.
- Die Projektplanung sah Meilensteine entlang den Schichten des Systems vor. Als Basis für die weitere Implementierung erfolgte nach dem Entwurf zunächst die Realisierung der außerhalb der Drei-Schicht-Architektur benötigten Basiskomponenten (Anbindung Persistenzrahmenwerk, Protokollierung, Synchronisation, Meldungswesen) und der Entwurf von Templateklassen für die zu realisierenden Komponenten der Drei-Schicht-Architektur. Als erste Schicht der Anwendung wurde darauf aufbauend die Datenhaltungsschicht implementiert. Der nächste Meilenstein sah die Entwicklung der Anwendungslogik vor und der letzte Projektschritt umfasste die Realisierung der Benutzerschnittstellen.
- Alle Projektmitglieder arbeiteten folglich parallel an der Implementierung derselben Architekturschicht. Die Zwischenergebnisse aller Beteiligten standen über ein System zur Versionsverwaltung (*CVS*) allen Entwicklern permanent zur Verfügung. Als verbindliche projektinterne Richtlinie wurde vereinbart, dass der im *CVS* verfügbare Stand in jeder Phase der Entwicklung kompilierfähig sein musste.

Da wie eingangs beschrieben die Entwicklung und Durchführung von Teststrategien kein expliziter Bestandteil des Praktikums war, wurde ohne nähere Planung durch diese Vorgehensweise parallel zur Implementierung implizit sofort mit dem Integrationstest begonnen. Ein isolierter Test von kleineren Teileinheiten des Systems war weder vorgesehen noch wurde er für notwendig erachtet. Dies führte in der Konsequenz dazu, dass weder bei Architekturentscheidungen in der Entwurfsphase noch während der Implementierungsphase Testbarkeitsaspekte Berücksichtigung fanden.

Hinzu kam noch, dass im Hinblick auf den Einsatz des Persistenz-Rahmenwerks sich die Bemühungen in erster Linie darauf richteten, eine funktionierende Anbindung der Datenbank zu erreichen. Für eine tiefergehende Beschäftigung, beispielsweise einem Vergleich verschiedener Alternativen zur Anbindung des Rahmenwerks an die Anwendung, blieb dabei keine Zeit.

Eine der Aufgaben des zweiten Teils dieser Arbeit (siehe Abschnitt 8.1) sollte die Erstellung von Testfällen für ausgewählte Teilkomponenten, die Refaktorisierung der Teilkomponenten zur Beseitigung von testkritischen Abhängigkeiten und die erneute vergleichende Erstellung von Testfällen auf Basis der veränderten Architektur sein. Um die Ermittlung des Aufwandes für die Erstellung einzelner Testfälle frei zu halten von Aufwänden für die Diskussion grundlegender systemweiter Testprobleme erschien es sinnvoll, diese Diskussion vor Beginn der Testfallerstellung zu führen.

Dies wird in den nächsten Abschnitten erfolgen. Dort werden einige zentrale Entwurfsprobleme und ihr möglicher Einfluss auf die Testbarkeit diskutiert. Zusätzlich zur Diskussion erfolgt an manchen

Stellen der Versuch, Lösungsvorschläge zur Vermeidung der erkannten Probleme zu geben. Die Umsetzung eines Teils der Lösungsvorschläge erfolgt während der Refaktorisierung von *SeminarIS* (siehe Kapitel 11 - *Entfernung testkritischer Abhängigkeiten durch Refaktorisierung*).

10.2 Entwurfsüberblick SeminarIS

Als Einstieg für die Auseinandersetzung mit der Architektur und dem Entwurf von SeminarIS dient das in der Entwurfsphase geplante, aber durch verschiedene Entwurfsfehler in dieser Form nicht verwirklichte Klassendiagramm der Anwendung. Die vorgesehenen Abhängigkeiten zwischen den einzelnen Komponenten sind durch Pfeile dargestellt:

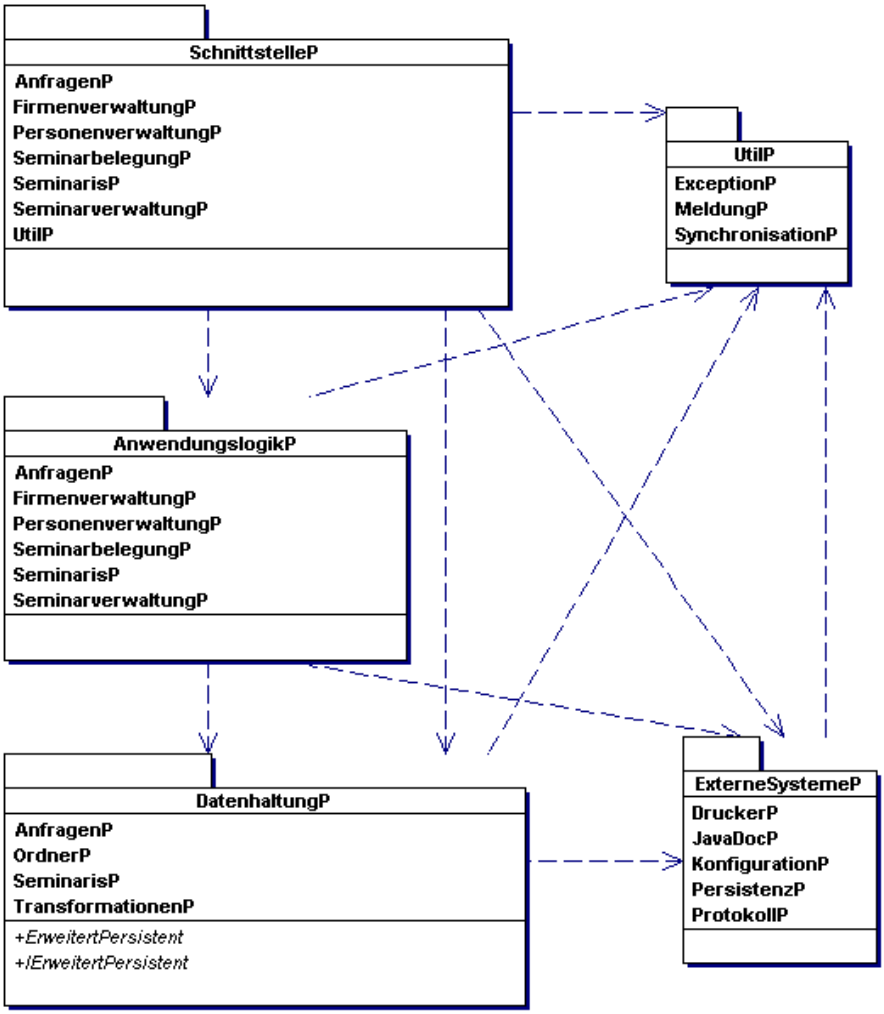


Abbildung 98: Paketdiagramm SeminarIS (Entwurfsphase)

In der Abbildung fehlen demzufolge alle während der Implementierungsphase eingefügten zusätzlichen Abhängigkeiten zwischen den dargestellten Paketen. Diese im Entwurf nicht vorgesehenen Abhängigkeiten werden beinahe ausnahmslos im Rahmen der Diskussion der Entwurfsprobleme ihre Erwähnung finden.

Einen groben Überblick über den Aufbau der Datenhaltungsschicht liefert das folgende Klassendiagramm:

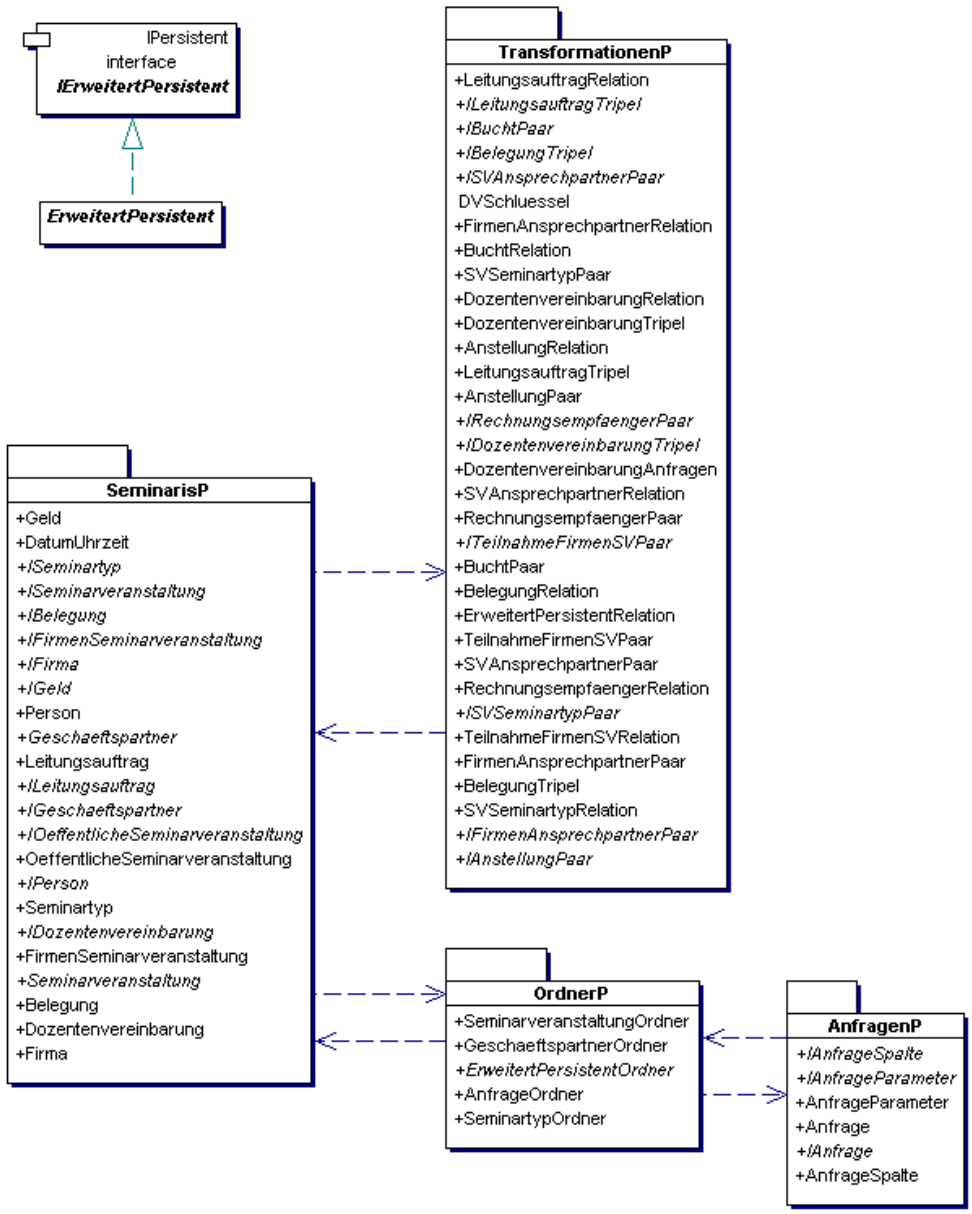


Abbildung 99: Paketdiagramm Datenhaltungsschicht

Das Paket `SeminarisP` umfasst die Domänenklassen und die Assoziationsklassen, die bereits aus dem Klassenmodell der Anforderungsspezifikation abgeleitet werden konnten. Im Paket `TransformationenP` sind die Relationen-, Paar- und Tripelklassen enthalten, die sich aus der Transformation der Assoziationen des Klassenmodells im Rahmen des Feinentwurfs ergaben. Im Paket `OrdnerP` sind die Ordnerklassen abgelegt, die den Zugriff auf die Instanzen der Echte Ganzes-Klassen bereitstellen **[**SWEII99**]**. Die in den Ordnerklassen enthaltene Funktionalität beschränkt sich dabei ausschließlich auf den Datenbank-Zugriff über die SQL-Abfrage-Schnittstelle des Persistenz-Rahmenwerks. Weitergehende Funktionalität für die Entitäten ist in den Domänenklassen realisiert.

Von den Klassen `IErweitertPersistent` und `ErweitertPersistent` sind alle persistenten Domänen- und Paarklassen abgeleitet. Sie erweitern die Klasse `IPersistent` des Persistenz-Rahmenwerkes um zusätzliche Funktionalität für den Umgang mit dem Persistenz-Rahmenwerk und beinhalten ferner eine Methode für den Zugriff zur Datenbank, die an die abgeleiteten Klassen über Vererbung bereit gestellt wird. Dieser Zugriffsmechanismus auf die Datenbank ist völlig analog in den Klassen `ErweitertPersistentOrdner`, von der alle Ordnerklassen abgeleitet sind, und `ErweitertPersistentRelation`, von der alle Relationenklassen erben, implementiert. Obwohl der Namensbestandteil „Persistent“ eine Verbindung zum Persistenz-Rahmenwerk vermuten lässt (insbesondere die Implementierung des Interfaces `IPersistent`), sind die letzten beiden ErweitertPersistent...-Klassen völlig unabhängig von der Persistenzthematik.

Das weitgehend eigenständige Paket `AnfragenP` realisiert Funktionalitäten für einen in der Anwendung integrierten Abfragegenerator.

Die Struktur der Anwendungslogik ergibt sich aus folgendem Klassendiagramm:

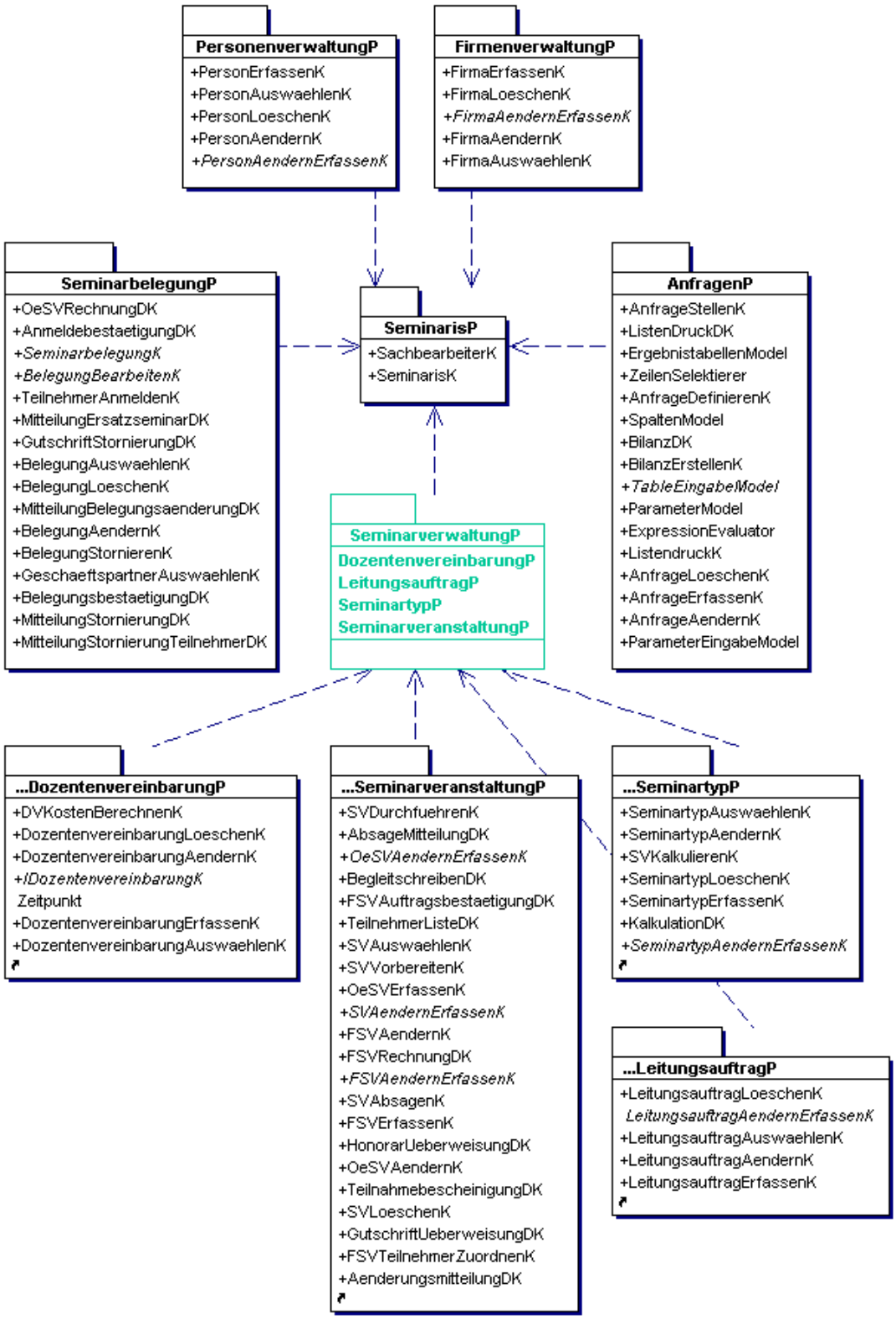


Abbildung 100: Paketdiagramm Anwendungslogik

Die vorhandenen Kontrollklassen orientieren sich an den durch die Anforderungsspezifikation vorgegebenen Anwendungsfällen. Einige der Kontrollklassen finden eine nochmalige Untergliederung durch das Paket `SeminarverwaltungP`. Da dieses Paket keine Klassen, sondern nur Pakete enthält, wurde es in der Darstellung ausgegraut.

Zentrale Instanz in der Kontrollschicht ist die Hauptkontrollklasse `SeminarisK`. Die in ihr realisierte Funktionalität beschränkt sich auf das Anbieten von statischen Variablen für den Zugriff auf die Basiskomponenten des Systems, wie beispielsweise Datenbank, Protokoll und Drucker.

10.3 Verletzung der Drei-Schicht-Architektur

10.3.1 Entwurfsproblem

Obwohl als Vorgabe die Entwicklung einer strikten Drei-Schicht-Architektur galt, wurde an drei Stellen aus der Datenhaltungsschicht direkt auf die übergeordnete Kontrollschicht zugegriffen:

- Die Domänenklasse `Dozentenvereinbarung` besitzt Abhängigkeiten zu den Klassen `DVKostenBerechnenK` und `SeminarisK` der Logikschicht.
- Die Transformationsklassen `DozentenvereinbarungAnfragen` und `DozentenvereinbarungTripel` greifen auf die Klasse `SeminarisK` zu.

Diese Abhängigkeiten durchbrechen folglich die hierarchische Ordnung der Schichtenarchitektur und machen damit den Austausch der höher liegenden Kontrollschicht nicht mehr möglich, ohne Veränderungen an der tiefer liegenden Datenhaltungsschicht vornehmen zu müssen.

Für die Kommunikation zwischen Benutzerschnittstelle und Anwendungslogik findet man analoge Verletzungen der Hierarchie. Aus folgenden Kontrollklassen erfolgt der Zugriff in die Schnittstellenschicht (auf ein statisches Attribut der Hauptschnittstellenklasse `SeminarisH`):

- `DVKostenBerechnenK`
- `DozentenvereinbarungAendernK`
- `DozentenvereinbarungAuswaehlenK`
- `DozentenvereinbarungErfassenK`
- `DozentenvereinbarungLoeschenK`

Betrachtet man unter diesem Gesichtspunkt die Drei-Schichten-Architektur von SeminarIS, so gibt das folgende Schaubild Aufschluss über die zyklische Vernetzung aller Schichten:

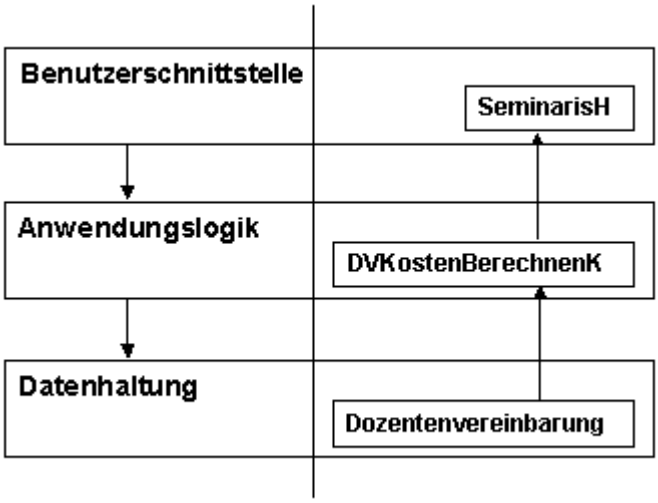


Abbildung 101: Invertierung der Drei-Schichten-Architektur in SeminarIS

Man sieht, dass durch die oben angeführten Verletzungen der Drei-Schichten-Architektur alle Schichten zyklisch miteinander verbunden sind. Lediglich der Umstand, dass der Aufruf der Anwendungsfälle in der Benutzerschnittstelle über einen in diesem Zusammenhang eigentlich nicht statthaften *Reflection*-Mechanismus erfolgt und dadurch ein Durchbrechen der zyklischen Abhängigkeiten erreicht wird, verhindert eine zyklische Vernetzung des nahezu gesamten Anwendungssystems. Implementiert man in der Schnittstellenschicht zur Veranschaulichung explizit alle Abhängigkeiten, die über den angesprochenen Reflection-Mechanismus ausgeblendet werden, so erhält man für *SeminarIS* folgende Projektmetriken:

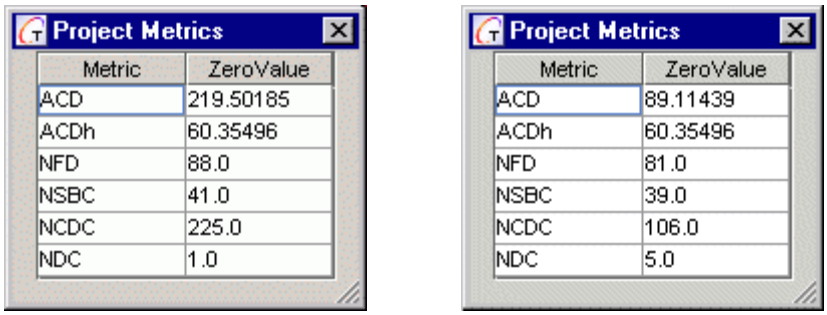


Abbildung 102: Vergleich der Projektmetriken, links ohne Reflection, rechts Originalprojekt

Man erkennt deutlich die unheilvollen Auswirkungen der durch die oben beschriebenen Abhängigkeiten ausgelösten Invertierung der Drei-Schicht-Architektur:

- Ohne Verwendung von Reflection wären 225 (!) Komponenten des Systems in einem einzigen gewaltigen Zyklus voneinander abhängig.
- Jede Komponente des Systems wäre im Durchschnitt von ca. 219 (!) anderen Komponenten direkt oder indirekt syntaktisch abhängig.

Interessant ist, dass die Abhängigkeitsmetriken nach dem Entfernen der *Reflection*-Mechanismen noch deutlicher auf dieses Entwurfsproblem hinweisen:

Abhängige Klasse	Funktionalität bereitstellende Klasse	dep_comp	is_feedb	is_intpk	rACD	rACDh	sub_hw	subedges	Typ der Abhängigkeit
Dozentenvereinbarung	DVKostenBerechnenK	1	0	1	50,3	9,9	11	9	create and access
SeminarisH	SachbearbeiterA	1	1	0	49,3	2,4	33	3	static access
SachbearbeiterA	AnfrageStellenAA	1	0	1	10,6	-	0	1	dep. to class
SeminarisK	SeminarisDatenbank	1	0	1	5,7	12,6	20	5	static access
SachbearbeiterA	BelegungAuswaehlenAA	1	0	1	5,4	-	0	1	dep. to class
AnfrageStellenAA	AnfrageStellenK	1	0	1	2,7	0,1	8	12	create and access
DruckDokumentDK	SeminarisK	1	0	1	2,5	2,9	100	4	static access
SachbearbeiterA	SVVorbereitenAA	1	0	1	1,9	-	0	1	dep. to class
BelegungAuswaehlenAA	BelegungStornierenAA	1	0	0	1,9	0,0	33	3	create and access
ErweitertPersistent	SeminarisDatenbank	1	0	1	1,7	4,4	100	1	static access
SVVorbereitenAA	SVVorbereitenK	1	0	1	1,6	0,0	10	10	create and access
TeilnehmerAnmeldenAA	TeilnehmerAnmeldenK	1	0	1	1,6	0,1	25	4	create and access
BelegungStornierenAA	BelegungStornierenK	1	0	1	1,6	0,1	50	2	create and access
SVAuswaehlenAA	FSVErfassenAA	1	0	0	1,6	0,5	33	3	create and access
BelegungAuswaehlenAA	BelegungAendernAA	1	0	0	1,6	0,0	50	2	create and access
SachbearbeiterA	SVDurchfuehrenAA	1	0	1	1,6	-	0	1	dep. to class
SVKurzS	Seminarveranstaltung	1	0	1	1,3	-	0	8	dep. to abstract class
FSVErfassenAA	FSVErfassenK	1	0	1	1,2	0,4	33	3	create and access
BelegungAendernAA	BelegungAendernK	1	0	1	1,2	0,1	50	2	create and access

Abbildung 103: Abhängigkeitsmetriken ohne Reflection-Aufrufe der Schnittstellenklassen

10.3.2 Auswirkungen auf die Testbarkeit

Im Modultest möchte man eine Komponente möglichst isoliert von allen anderen Komponenten des Systems testen. Hängt wie in *SeminarIS* eine Klasse im Durchschnitt von knapp 90 anderen Klassen ab (Metrik ACD), so führt dies zu folgenden Problemen:

- Ohne den Einsatz von Stellvertretern müssen für den Modultest alle abhängigen Klassen kompilierfähig im System vorhanden sein. Dies reduziert die möglichen Testreihenfolgen und führt unter Umständen zu Verzögerungen in der Testdurchführung.
- Jede dieser Klassen muss in wenigstens einem Testfall des Moduls kompiliert und unter Umständen auch geladen und instantiiert werden. Dies erhöht den Zeitaufwand für die Testdurchführung erheblich.
- Als Ursprung von Fehlern kommen grundsätzlich alle beteiligten Komponenten in Betracht, was die Lokalisierung von Fehlern deutlich erschwert.

Eine Möglichkeit, diese Probleme zu beseitigen oder wenigstens zu reduzieren, besteht in der Verwendung von Stellvertretern (*Stubs* oder *Mocks* [Link01]) für die benötigten Klassen. Dass von den durchschnittlich 90 benötigten Komponenten zwei Drittel über fest verdrahtete Abhängigkeiten verbunden sind (Metrik ACDh), verhindert jedoch für diese Klassen die Anwendung dieser Strategie. Diese Komponenten können nur dann durch Stellvertreter ersetzt werden, wenn Änderungen an der Implementierung der abhängigen Klasse vorgenommen werden.

Eine weitere Verschärfung der Testprobleme ergibt sich durch Abhängigkeitszyklen. Alle an einem Zyklus beteiligten Komponenten können nur zusammen getestet werden. Möchte man die Zyklen aufbrechen, so müssen zusätzliche Stellvertreter implementiert werden. Dies kann jedoch auch nicht verhindern, dass die Testfallerstellung durch Abhängigkeitszyklen erheblich an Komplexität gewinnt.

Ähnliche Probleme folgen auch für den Integrationstest. Dort sind Strategien zu bevorzugen, bei denen die Integration der Module inkrementell verläuft. Im Idealfall werden dabei „gut“ getestete Komponenten in kleinen Schritten dem System hinzugefügt. Dies bringt folgende Vorteile:

- Bei geeigneter Wahl der Integrationsstrategie (z.B. *bottom up*) kann weitgehend auf den Einsatz von Stellvertretern und den zur Erstellung nötigen hohen Aufwand verzichtet werden.
- Neu hinzugekommene Fehler können in unmittelbarer Nähe der zuletzt integrierten Komponente vermutet und daher schneller lokalisiert werden.

Bei Integrationsstrategien, die den sofortigen gleichzeitigen Test einer großen Anzahl von Komponenten vorsehen (Big Bang), liegt die Schwierigkeit darin, aufgetretene Fehler zu lokalisieren. Außerdem kann bei diesem Vorgehen nur schwerlich eingeschätzt werden, welche Fehler durch andere Fehler bedingt werden.

Inkrementelle Strategien werden durch hierarchische Architekturen unterstützt **[**Lako96**]**. Liegt wie bei *SeminarIS* durch die zyklische Struktur nahezu aller Anwendungskomponenten eine Auflösung sämtlicher Hierarchieebenen vor, so kann ein sukzessiver Test von immer größer werdenden Teilmodulen nur mit erheblichem Aufwand (z.B. durch weitreichenden Einsatz von Teststellvertretern) erreicht werden. In diesen Fällen wird man geneigt sein, unter Inkaufnahme der damit verbundenen Nachteile auf eine andere Integrationsstrategie (Big Bang) auszuweichen.

10.4 Zugriff auf Kontrollschicht durch Basiskomponenten

10.4.1 Entwurfsproblem

Ein ähnliches Problem wie im vorhergehenden Abschnitt stellt sich auch durch den Zugriff auf die Kontrollschicht durch die Basiskomponenten der Pakete `UtilP` und `ExterneSystemeP`, die sich außerhalb der Drei-Schicht-Architektur befinden:

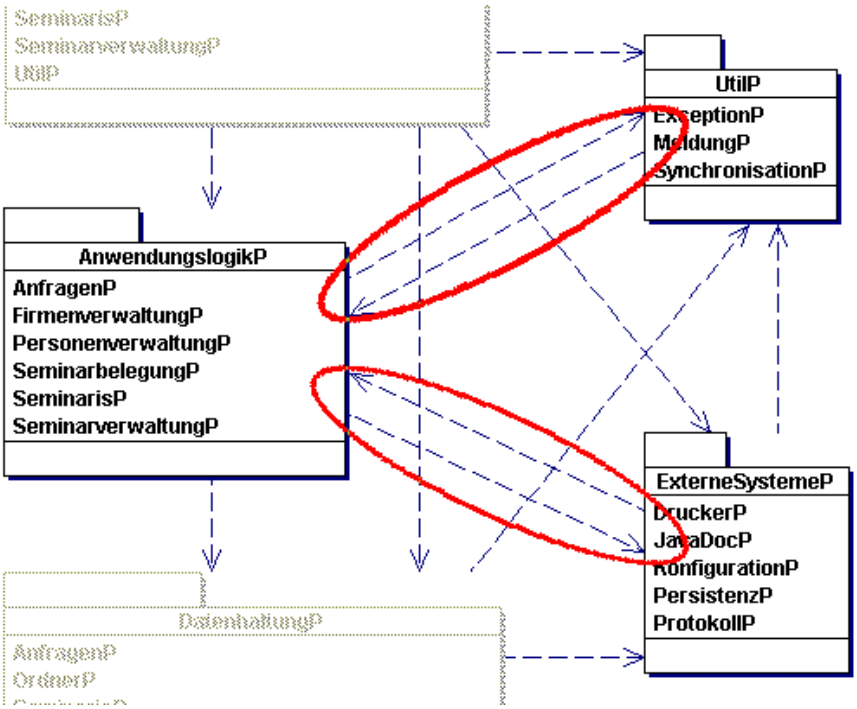


Abbildung 104: Zugriff auf Kontrollschicht aus externen Modulen (Ausschnitt aus SeminarIS)

Damit verlieren diese Pakete und die in ihnen enthaltenen Komponenten ihre Unabhängigkeit von der Anwendung. Erfolgt der Austausch oder eine Veränderung der Kontrollschicht, so müssen die Teile der Basiskomponenten, in denen der Zugriff auf die Kontrollschicht erfolgt, gegebenenfalls angepasst werden.

Die Zugriffe erfolgen auf die globalen Variablen `SeminarisK.Protocol` und `SeminarisK.Config` der Hauptkontrollklasse. Bemerkenswert hieran ist, dass es sich bei diesen globalen Variablen der Kontrollschicht um Instanzen anderer Basiskomponenten handelt. Damit liegt eine unnötige Inanspruchnahme der Kontrollschicht vor und demzufolge auch eine sehr einfach zu entfernende Abhängigkeit. Die beschriebenen Zugriffe verteilen sich auf folgende Pakete:

- Zugreifende Klassen im Paket `ExterneSystemeP.ProtokollP`:
  - `ProtokollDateiA`
  - `LogProtokollDateiA`
  - `DebugProtokollDateiA`

- Zugreifende Klassen im Paket `UtilP.MeldungP`:
  - `MeldungsAnzeigeA`

- Zugreifende Klassen im Paket `ExterneSystemeP.DruckerP`:
  - `DruckDokumentDK`
  - `BriefDK`

Aufgrund der Implementierung von `SeminarisK` entsteht durch diese Zugriffe eine transitive Abhängigkeit der aufgeführten Klassen von der Datenbank der Anwendung und aufgrund der Implementierung der Datenbank (siehe nachfolgenden Abschnitt 10.5) darüber hinaus auch noch die Abhängigkeit von der Datenhaltungsschicht.

Die beschriebenen Probleme resultieren letztlich aus dem ungeschickt realisierten Versuch, den systemweiten Zugriff auf globale Basiskomponenten zu ermöglichen. Eine detaillierte Diskussion dieses Problems findet sich in Abschnitt 10.6. Dort werden auch Möglichkeiten für eine alternative Implementierung des Zugriffs vorgestellt.

10.4.2 Auswirkungen auf die Testbarkeit

Die beschriebenen Zugriffe auf die Kontrollschicht führen grundsätzlich ebenfalls zu den in Abschnitt 10.3.2 erläuterten Testproblemen.

Zudem führt die transitive statische Abhängigkeit der Basiskomponenten zur Datenbank zu den in Abschnitt 10.5.2 beschriebenen Problemen.

10.5 Zugriff auf Datenhaltung durch Basiskomponente

10.5.1 Entwurfsproblem

Analog zu den bereits beschriebenen Problemen wird die hierarchische Struktur der Anwendung durch den fest verdrahteten Zugriff auf acht Datenhaltungsklassen aus der Klasse `SeminarisDatenbank` (Paket `ExterneSystemeP.PersistenzP`) verletzt:

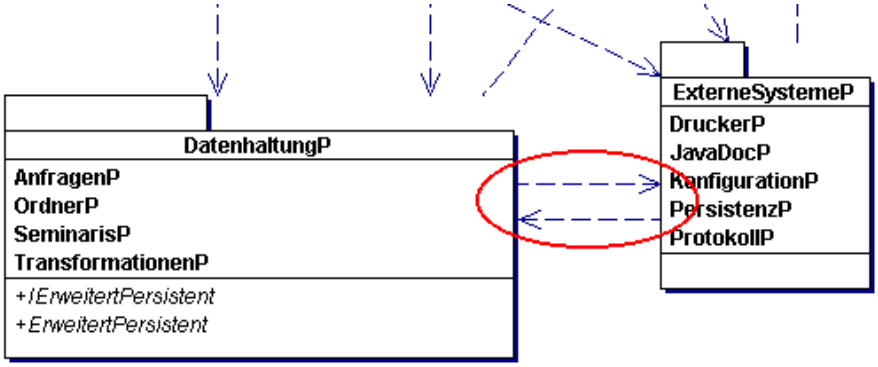


Abbildung 105: Zugriff auf Datenhaltung aus externem Persistenzmodul (Ausschnitt aus SeminarIS)

Der Entwurfsfehler entstand als Folge der Vorgabe der Anforderungsspezifikation, dass für bestimmte Entitäten ein eindeutiger, für den Anwender sichtbarer und im Geschäftsverkehr auch Verwendung findender Bezeichner vom System generiert werden sollte. Für die Erzeugung und Verwaltung dieser eindeutigen Bezeichner wurde die folgenschwere Entscheidung getroffen, diesen Mechanismus in der Hauptkontrollklasse und der Datenbankklasse anzusiedeln:

- `SeminarisK` bietet statische synchronisierte Methoden an, mit denen jeweils der nächste zu vergebende Bezeichner abgerufen werden kann (z.B. `getNextKundennummer()`).
- Die Initialisierung der Bezeichner erfolgt beim Start der Anwendung, indem der höchste bisher in der Datenbank vergebene Bezeichner ermittelt wird.
- Für diese Initialisierung bietet `SeminarisDatenbank` Methoden an, die durch Datenbankabfragen diese Werte liefern (z.B. `getHoechsteKundennummer()`).
- Diese Methoden werden beim Laden der Klasse `SeminarisK` zur Initialisierung der Bezeichner aufgerufen.

Die Implementierung der Methoden `getHoechste...nummer()` zur Ermittlung der zuletzt vergebenen Bezeichner in der Klasse `SeminarisDatenbank` verletzt das Prinzip der Kohäsion. Unter dem Gesichtspunkt des starken logischen Zusammenhangs der Funktionalität von Klassen sollten die Methoden an der Stelle angesiedelt sein, von der aus die Verwaltung und der Zugriff auf die jeweilige Domäne erfolgt. Dies gilt völlig analog auch für die Methoden `getNext...nummer()` in der Hauptkontrollklasse `SeminarisK`.

10.5.2 Auswirkungen auf die Testbarkeit

Der Zugriff aus der Datenhaltung in die Kontrollschicht (siehe Abschnitt 10.3.1) in Verbindung mit dem hier beschriebenen Zugriff der Datenbank in die Datenhaltung führt zu der in Abschnitt 9.2.2 veranschaulichten Verstärkung der zyklischen Vernetzung in der Datenhaltungsschicht. Insoweit treten auch hier die in Abschnitt 10.3.2 erläuterten Testprobleme auf.

Im Detail ergeben sich darüber hinaus folgende Probleme: Wird in einem Testfall auf die Klasse `SeminarisK` zugegriffen, erfolgt die Initialisierung der dort angesiedelten statischen Variablen für die eindeutigen Bezeichner. Dies führt über den Aufruf der Initialisierungsmethoden `getHoechste...nummer()` der *Singleton*-Klasse `SeminarisDatenbank` bereits zu Abfragen auf die Datenbank. Sind die zu den Bezeichnern gehörenden Domänenklassen zu diesem Zeitpunkt der Datenbank noch nicht bekannt, werden vor jeder Abfrage vom Persistenz-Rahmenwerk zudem noch die Proxy-Klassen für diese Domänenklassen erzeugt und kompiliert.

Dieses Problem wirkt sich deshalb extrem negativ auf die Testbarkeit aus, weil aufgrund der statischen Abhängigkeiten dieser Automatismus der Datenbank-Abfrage und Proxygenerierung beim Test von

- nahezu allen Klassen der kompletten Kontrollschicht (`SeminarisK` wird fast in jeder Kontrollklasse für Transaktionssteuerung benötigt)
- und fast allen Klassen der Schnittstellenschicht (die Schnittstellenklassen verwenden die Hauptschnittstellenklasse `SeminarisH`, in der obligatorisch die Erzeugung von `SeminarisK` erfolgt)

gestartet wird. Aufgrund der statischen Art der Abhängigkeit lässt sich ohne Änderung der Implementierung auch in der Testumgebung kein anderes Verhalten erreichen (siehe auch Abschnitt 10.6.2).

Auch für Klassen, in denen mit der tatsächlichen Implementierung der Datenbank getestet werden soll (z.B. Ordnerklassen, da hier die syntaktische und semantische Korrektheit der verwendeten SQL-Anweisungen nur im Zusammenspiel mit dem Persistenz-Rahmenwerk und der Datenbank getestet werden kann), ergeben sich Probleme. Für diese Tests muss eine kompilierfähige Implementierung der Domänenklassen vorhanden sein, deren zuletzt vergebene eindeutige Bezeichner über die Datenbankklasse ermittelt werden.

10.6 Zugriff auf globale Basiskomponenten

10.6.1 Entwurfsproblem

Ein zentrales Problem, das auch bereits in den vorhergehenden Abschnitten deutlich wurde, ist die Realisierung eines Zugriffs auf einzelne systemweit benötigte Komponenten. Für einen globalen Zugriff vorgesehen sind in SeminarIS u.a. folgende Komponenten:

- Datenbank inklusive Transaktionen (Klasse `SeminarisDatenbank`).
- Protokollierung (Paket `ProtokollP`).
- Synchronisation (Paket `SynchronisationP`).
- Konfiguration (Paket `KonfigurationP`)
- Hauptanwendungsfenster (Klasse `SachbearbeiterA`).

Der Zugriff auf diese Komponenten erfolgte auf unterschiedliche Weise, beispielsweise durch Verwendung von statischen Variablen in den Hauptklassen der Anwendungsschichten: `SeminarisK.Database`, `SeminarisK.Protocol`, `SeminarisK.Sync`, `SeminarisK.Config` und `SeminarisH.MainWindow`. Anders als in der Kontrollschicht wurde der Zugriff auf die Datenbankinstanz in der Datenhaltung über die Implementierung der Methode `getDb()` in den Klassen `ErweitertPersistent...` realisiert. Über Vererbung wird die Funktionalität dort an die Datenhaltungsklassen weitergegeben. Darüber hinaus werden an weiteren Stellen der Anwendung redundant lokale Variablen für die Datenbank definiert oder es erfolgt der direkte Zugriff auf die *Singleton*-Instanz der Datenbankklasse.

Schon die Tatsache, dass in den Anwendungsschichten unterschiedliche Konzepte zur Bereitstellung der Datenbank implementiert sind, zeigt die Unsicherheit, wie der Zugriff auf globale Ressourcen am günstigsten erfolgen sollte. Wie sich in den Abschnitten 10.3 bis 10.5 gezeigt hat, scheinen die angeführten Arten der Realisierung des Zugriffs die Gefahr in sich zu bergen, durch versehentliche fehlerhafte und unkoordinierte Zugriffe hierarchische Strukturen von Anwendungssystemen zu beeinträchtigen, wenn nicht sogar zu verhindern. Darüber hinaus führt die zentrale Bündelung dieser Basiskomponenten innerhalb einer Klasse (`SeminarisK`) dazu, dass jede Komponente, die auch nur von einer einzigen Basiskomponente abhängig ist, Abhängigkeiten von allen weiteren

Basiskomponenten erfährt. In Abschnitt 11.2 erfolgt eine detaillierte Beschreibung möglicher alternativer Zugriffsstrategien und deren Diskussion und Bewertung aus Sicht der Testbarkeit.

Exkurs:

Nur am Rande erwähnt, aber in diesem Zusammenhang sicherlich hochinteressant, ist das technische Konzept der Firma Sun Microsystems mit ihrer Java™ 2 Enterprise Edition (J2EE) als umfangreicher Baukasten, auf den bei der Realisierung von Anwendungssystemen zurückgegriffen werden kann. Die Grafik stellt das Architekturmodell der J2EE dar:

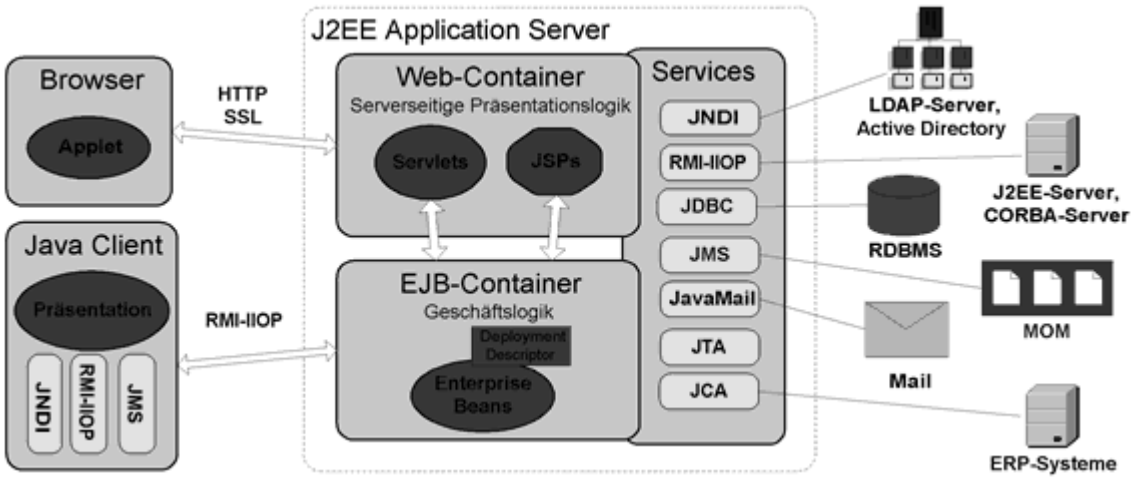


Abbildung 106: Überblick J2EE-Architektur [Quelle: www.movingobjects.de]

Bei den vom Anwendungsentwickler zu realisierenden Teilen handelt es sich um die dunkel hinterlegten Komponenten im linken und mittleren Teil der Grafik (z.B. Präsentation, JSPs). Das Bemerkenswerte in diesem Zusammenhang ist, dass durch Bereitstellung zahlreicher technische Basisdienste wie Namensdienst (*JNDI*), Zugriff auf Datenbanken (*JDBC*, *JCA*), deklaratives Transaktions-Management (*JTA, JCA*) und Persistenz (*EJB*), der Entwickler gerade von den in diesem Abschnitt beschriebenen Aufgaben und Schwierigkeiten Entlastung finden soll.

10.6.2 Auswirkungen auf die Testbarkeit

Die Probleme, die durch einen versehentlichen bzw. fehlerhaften Zugriff auf eine der Basiskomponenten entstehen können, wurden bereits geschildert (Abschnitte 10.3 bis 10.5).

Darüber hinaus treten weitere Probleme auf:

Der Zugriff auf die Datenbank erfolgt in allen Varianten fest verdrahtet. Dies ist auf die Realisierung der Klasse `SeminarisDatenbank` als Einzelstück (*Singleton*) zurückzuführen. Sowohl in `SeminarisK` als auch in den `ErweitertPersistent`-Klassen erfolgt die Erzeugung und Bereitstellung der Datenbankinstanz durch Aufruf der statischen Methode `SeminarisDatenbank.getEinzigeInstanz()`. Durch die Abhängigkeit der Kontrollklassen von `SeminarisK` und der Ableitung der Datenhaltungsklassen von `ErweitertPersistent` (vergleiche Abschnitt 10.5.2) können in der Kontrollschicht und der Datenhaltungsschicht keine Tests in Isolation von der Datenbank durchgeführt werden. Dies kann nur über das Verändern des zu testenden Codes oder das Ersetzen der Klassen `SeminarisK`, `ErweitertPersistent` oder `SeminarisDatenbank` durch eine geänderte Implementierung erreicht werden.

Grundsätzlich stellt sich dasselbe Problem auch für die anderen Basiskomponenten dar. Auch der Zugriff auf die Konfigurations- und Protokollkomponente ist nach dem Entwurfsmuster *Singleton* realisiert, so dass auch hier keine Konfigurierbarkeit der Anwendung für Testzwecke möglich ist. Im Übrigen drängt sich der Verdacht auf, dass allgemein bei der Entscheidung für den Einsatz dieses Entwurfsmusters die Möglichkeit des globalen Zugriffs über die statische `getInstance()`-Methode womöglich häufig die bedeutendere Rolle spielt als die Absicht zur Beschränkung der erzeugbaren Instanzen.

10.7 Realisierung der Domänenklassen als Fassaden

10.7.1 Entwurfsproblem

Einen Überblick über den Zugriff aus der Kontrollschicht auf die Datenhaltung gibt das nachfolgende Paketdiagramm. Auf den ersten Blick wird die Fassadenfunktion der Domänenklassen deutlich (Paket `Datenhaltung.SeminarisP`). Im Diagramm sind alle Abhängigkeiten ausgeblendet, die eine Verletzung der Mehrschichtarchitektur und des Fassadenmusters darstellen:

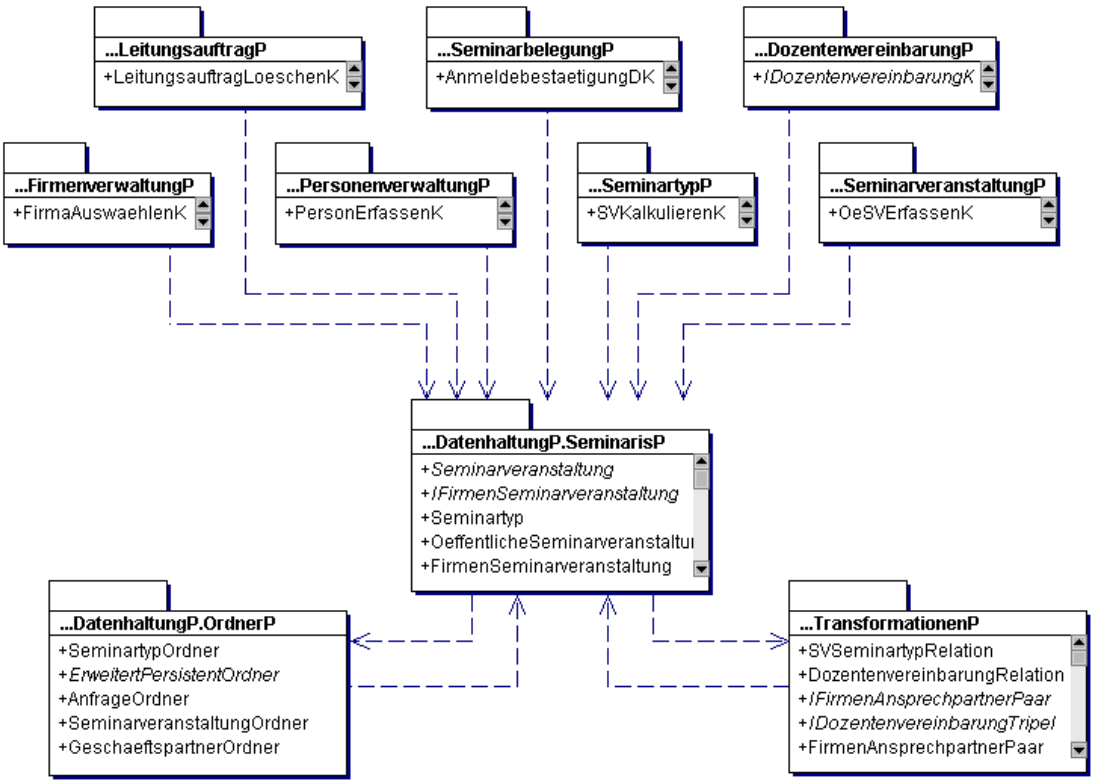


Abbildung 107: Zugriff auf Datenhaltung mit Fassadenmuster

Der obigen Architektur und der daraus resultierenden Effekte liegt folgende Design-Entscheidung zugrunde:

- Auf die Datenhaltungsschicht sollte exklusiv über die originären Entitätsklassen (Domänenklassen) zugegriffen werden.
- Die Domänenklassen (`Seminartyp`, `Firma`, ...) beinhalten also zu einem gewissen Teil die Aufgaben einer nach dem Entwurfsmuster *Fassade* realisierten Klasse (neben Ihrer eigentlichen fachlichen Funktionalität).

- Wird demzufolge in der Anwendungslogik oder in anderen Datenhaltungsklassen Funktionalität aus den Ordner- oder Relationenklassen benötigt, so werden die Methodenaufrufe von den Domänenklassen entgegen genommen und an die jeweils zuständigen Relationen- und Ordnerklassen durchgereicht.
- In den meisten Fällen ist dabei keine zusätzliche Funktionalität in der Domänenklasse angesiedelt, d.h. die Aufgabe der Domänenklassenmethode beschränkt sich ausschließlich auf das Durchreichen des Methodenaufrufs.

Dieses Design entsprach nicht den Empfehlungen aus [SWEII99] und wurde demzufolge auch kontrovers diskutiert. Letztlich setzte sich innerhalb der Entwicklergruppe folgende Argumentation durch:

- Entitätsklassen lassen sich in objektorientierten Sprachen unmittelbar aus den Entwurfsmodellen ableiten.
- Assoziationen hingegen müssen erst in „implementationsfähige" Entwurfsklassen transformiert werden.
- Hierfür existieren unterschiedliche Transformationsvarianten, von denen es nicht immer „die Beste" gibt.
- Deshalb sollte dieser möglicherweise eher veränderliche Teil der Implementierung der Datenhaltung vor der Kontrollschicht verborgen bleiben.
- Hierfür bietet sich obligatorisch das Entwurfsmuster *Fassade* an.

Das eben angesprochene ausschließliche Durchreichen des Methodenaufrufs wird aus folgendem Programmauszug der Domänenklasse `Firma` deutlich:

```
public static Iterator getAlle() throws DatenbankAusnahme {
   return iter = GeschaeftspartnerOrdner.getEinzigeInstanz().getFirmen();
}
```

Muss die von einer Ordner- oder Relationenklasse im Rahmen einer Datenbankabfrage ermittelte Ergebnismenge noch aufbereitet werden, so ist diese zusätzliche Logik nicht mehr in der Ordner- oder Relationenklasse, sondern bereits in der Domänenklasse realisiert, wie nachstehendes Beispiel aus der Domänenklasse `Firma` veranschaulicht:

```
public static Collection getAlleSchluessel() throws DatenbankAusnahme {
   Iterator iter = getAlle();
   Vector ergebnisMenge = new Vector();
   while (iter.hasNext()) {
      ergebnisMenge.add(((IFirma)iter.next()).getSchluessel());
   }
   return ergebnisMenge;
}
```

Der ausschließliche Zugriff über die Domänenklassen auf die Ordner- und Relationenklassen wurde nur an einigen wenigen Stellen durchbrochen (ohne funktionale Notwendigkeit, sondern lediglich aus Unwissenheit, Unachtsamkeit oder an ganz wenigen Stellen auch aus Performanzgründen).

Im Zusammenhang mit der beschriebenen Verletzung der Drei-Schichten-Architektur (Abschnitt 10.3) fällt auf, dass ein Auslöser in der Fassadenfunktion der Domänenklassen vermutet werden kann. Die Klasse `Dozentenvereinbarung` bietet in ihrer Schnittstelle eine Methode an, für deren Implementierung nennenswerte Logik zu realisieren ist. Um dies erfüllen zu können, reicht die Entitätsmethode den Aufruf auch an dieser Stelle im Stile einer Fassade weiter:

```
public IGeld getKosten(String vonDatum, String vonZeit,
             String bisDatum, String bisZeit) throws DatenbankAusnahme
{
   DVKostenBerechnenK dVKBK = new DVKostenBerechnenK();
   dVKBK.setBisTag(bisDatum);
   dVKBK.setBisUhrzeit(bisZeit);
   dVKBK.setVonTag(vonDatum);
   dVKBK.setVonUhrzeit(vonZeit);
   dVKBK.setStundensatz(stundensatz);
   dVKBK.setTagesarbeitszeit("8");
   return dVKBK.berechneKosten();
}
```

Dieses Mal allerdings nicht an die Methoden der Ordner- und Relationenklassen, sondern es erfolgt der Aufruf einer in der Kontrollschicht angesiedelten Methode. Dies führt zu der Abhängigkeit mit dem höchsten rACD-Wert des ganzen Systems (siehe Abschnitt 9.2.1). Möglicherweise wäre die Abhängigkeit nicht entstanden, wenn bei der Realisierung von *SeminarIS* auf die Fassadenfunktionalität in der Datenhaltung verzichtet worden wäre und zudem der strikte Grundsatz gegolten hätte, die Domänenklassen frei von jeglicher Geschäftslogik zu halten.

10.7.2 Auswirkungen auf die Testbarkeit

Beschränkt sich die Aufgabe der Domänenklassen-Methode ausschließlich auf das Durchreichen des Methodenaufrufs, so könnte man versucht sein, auf den Test der Methode zu verzichten, da die eigentliche Funktionalität bereits in der Ordner- oder Transformationenklasse getestet wurde.

Leider führt die Entscheidung zugunsten einer Fassade im Verbund mit dem Persistenz-Rahmenwerk zu gravierenden Problemen beim Durchreichen von Methodenaufrufen von den Domänenklassen an die transformierten Assoziationsklassen. Das Problem taucht auf, wenn sich die Domänenklasse beim Durchreichen des Methodenaufrufes selbst als Parameter übergeben muss.

Beispiel (Domänenklasse `Firma`):

```
public Iterator getAlleAnsprechpartner() throws DatenbankAusnahme {
   return
      // naive Implementierung - führt zu Programmfehler !!!
      FirmenAnsprechpartnerRelation.getEinzigeInstanz().getPersonen( this);
}
```

Laut Signatur erwartet die aufgerufene `getPersonen()`-Methode der Assoziationsklasse `FirmenAnsprechpartnerRelation` als Parameter ein beliebiges Objekt vom Typ `IFirma`. Die obige `this`-Instanz erfüllt diese Vorgabe (`Firma implements IFirma`).

Über einen weiteren Zwischenschritt wird letztlich folgende Methode aufgerufen:

```
private Iterator getFirmenAnsprechpartnerPaar( IFirma f)
                                               throws DatenbankAusnahme {
    try {
       Abfrage abfrage = new Abfrage( FirmenAnsprechpartnerPaar.class);
       StringBuffer sb = new StringBuffer();
       sb.append( "SELECT FirmenAnsprechpartnerPaar.ID FROM ");
       sb.append( "FirmenAnsprechpartnerPaar");
       sb.append( " WHERE ");
       sb.append( "firma = ?");
       Tupel tupel = new Tupel();
       tupel.setze( "firma", f);
       abfrage.setzeAnweisung( sb.toString());
       abfrage.setzeTupel( tupel);
       Iterator iter = getDb().abfrageAusfuehren( abfrage); /* # */
      ....
}
```

Bei der beschriebenen Implementierung handelt es sich beim Methodenparameter um ein Objekt vom Typ `Firma`. Dies führt an der gekennzeichneten Stelle (#) zu einer vom Persistenz-Rahmenwerk geworfenen Ausnahme. Der Grund hierfür ist, dass in der abgefragten Tabelle der Datenbank (`FirmenAnsprechpartnerPaar`) als Referenz auf eine Firma die Objekt-ID eines vom Persistenz-Rahmenwerk verwalteten Proxyobjektes gespeichert ist. Wird wie oben eine Instanz der Domänenklasse übergeben, so wird an dieser Stelle vom Rahmenwerk versucht, die übergebene Firmeninstanz in ein Proxyobjekt zu casten, um die Objekt-ID auszulesen. Dieser Cast führt zum Werfen der Ausnahme.

Um den Fehler zu beheben, wurde die Superklasse aller Domänenklassen, die Klasse `ErweitertPersistent`, um eine Methode `getThis()` erweitert, die zu einer Domäneninstanz das zugehörige Proxyobjekt erneut ermittelt. Dies geschieht wie folgt:

- Bestimmung des Stringschlüssel der Instanz (dieser Stringschlüssel wird zur Anzeige in allen Auswahlmasken verwendet und ist für alle Domänen eindeutig),
- Lesen aller Datensätze vom Domänentyp aus der Datenbank,
- Vergleich aller Datensätze mit dem Stringschlüssel der Instanz (da dieser Stringschlüssel im Unterschied zu den eindeutigen Bezeichnern nicht in der Datenbank gespeichert ist, kann keine konkrete Abfrage auf diesen Schlüssel formuliert werden).

Zur Vermeidung der oben beschriebenen Ausnahme müssen also (obwohl die Ergebnisinstanz bereits im Zugriff war) nochmals sämtliche Datensätze einer Domänentabelle aus der Datenbank gelesen werden. Dieser redundante lesende Zugriff auf die Datenbank wirkt sich sehr negativ auf die Performanz des Systems aus (teilweise Verdopplung der Antwortzeiten).

Die korrekt funktionierende Implementierung der durchreichenden Methode der Domänenklasse lautete schließlich wie folgt:

```
public Iterator getAlleAnsprechpartner() throws DatenbankAusnahme {
   return FirmenAnsprechpartnerRelation.
                   getEinzigeInstanz().getPersonen( (IFirma)getThis());
}
```

Der Zugriff auf die Methoden der Ordnerklassen bleibt vom oben beschriebenen Problem unberührt. Der Grund hierfür ist, dass in diesen Fällen von den Domänenklassen ausschließlich Parameter mit primitiven Datentypen (`String`, `Date`) als SQL-Abfrageparameter an die Ordnerklassen durchgereicht werden. Unabhängig davon ist das bei den Transformationenklassen geschilderte Problem Grund genug, um auch beim Durchreichen von Methoden an die Ordnerklassen Testfälle vorzusehen.

Dies führt zu folgenden negativen Auswirkungen auf die Testaktivitäten:

- Der Test der Fassaden-Methoden darf nicht entfallen, damit auch Fehler im Zusammenwirken mit dem Persistenz-Rahmenwerk entdeckt werden können.
- Hierdurch entstehen letztlich redundante Testfälle für alle durchgereichten Methoden. Neben dem Test der Methode in der Assoziationsklasse muss in der Domänenklasse erneut die erwartete Funktionalität getestet werden. Der ohnehin schon zusätzliche Implementierungsaufwand aufgrund des gewählten Entwurfsmuster führt also darüber hinaus zu einer Verdopplung des Testaufwands.
- Der Testaufwand ist noch dazu verhältnismäßig hoch. Müssen als Parameter persistente Objekte übergeben werden, so sind diese im Setup der Testfälle zu erzeugen. Dazu ist der Zugriff auf das Persistenz-Rahmenwerk und die Datenbank erforderlich, da die Implementierung eines Stellvertreters auf Basis der deklarierten Interfaces nicht ausreichend ist (siehe oben).

Würde man die Design-Entscheidung rückgängig machen (also den direkten Zugriff auf die Ordner- und Transformationenklassen erlauben und die Domänenklassen von jeglicher Logik befreien), so könnte der Testaufwand also in beträchtlichem Umfang reduziert werden.

10.8 Erzeugung der Schnittstellenklassen

10.8.1 Entwurfsproblem

Unter den Abhängigkeiten mit den höchsten rACD-Werten befinden sich insgesamt fünf Abhängigkeiten, die durch das Erzeugen und den Aufruf einer Auswahlschnittstellenklasse aus einer Schnittstellenklasse eines anderen Anwendungsfalls entstehen (siehe Abhängigkeiten 9.2.3, 9.2.5, 9.2.6, 9.2.7 und 9.2.9). In allen Fällen liegt als funktionale Anforderung zugrunde, bei der Erfassung oder Änderung einer Entitätsinstanz in einem Anwendungsfall ein mit dieser Instanz verbundenes Objekt bearbeiten oder auswählen zu können.

Ausgangspunkt für den Aufruf aller Anwendungsfälle ist das *SeminarIS*-Hauptanwendungsfenster, das durch die Klasse `SachbearbeiterA` implementiert wurde. Der Einstieg erfolgt bei allen Anwendungsfällen über das *SeminarIS*-Hauptmenü durch Aufruf einer anwendungsfallspezifischen Auswahlsmaske, die es gestattet, bereits gespeicherte Instanzen zur Bearbeitung auszuwählen oder alternativ auch Neuerfassungen vorzunehmen:

Abbildung 108: Obligatorische Auswahlmaske für alle Firmen-Anwendungsfälle

Beim Aufruf über das Hauptmenü erfolgt die Erzeugung der Schnittstellenklassen über Reflection-Mechanismen (siehe auch Abschnitt 10.3). Der Grund für die Verwendung von Java-Reflection war, unabhängig von der Fertigstellung der Anwendungsfälle durch die einzelnen Gruppenmitglieder, zu jedem Zeitpunkt die Kompilierfähigkeit des Systems auf jedem der verteilten Entwickler-Arbeitsplätze zu gewährleisten. Obwohl die Verwendung des Reflection-Mechanismus zu keinerlei Problemen führte, handelt es sich um eine nicht gerechtfertigte Entwurfsvariante, da die Verwendung von Reflection-Mechanismen grundsätzlich auch Nachteile mit sich bringt (z.B. Verringerung der Typsicherheit). Korrekterweise hätte als Lösung die Implementierung von Stellvertretern für noch nicht fertige Klassen erfolgen müssen, auch wenn damit ein etwas höherer Aufwand verbunden gewesen wäre. Die Testbarkeit des Systems wurde durch diese Entscheidung jedoch nicht beeinträchtigt.

Ohne Reflectionmechanismen werden die Auswahlschnittstellenklassen hingegen im Rahmen der Bearbeitung und Erfassung von Datensätzen aufgerufen, um eine über eine Assoziation verbundene Entitätsinstanz auszuwählen. Durch diesen Aufruf der Auswahlmasken aus anderen Anwendungsfällen entstanden die eingangs beschriebenen Abhängigkeiten. Grundsätzlich führt somit jeder Anwendungsfall, der über einen *exceptional-* oder *alternative-flow* in einen anderen Anwendungsfall verzweigt, zur Abhängigkeit der jeweiligen Schnittstellenklassen. Verlaufen die *flows* wechselseitig in beide Richtungen, so entstehen zudem zyklische Abhängigkeiten.

10.8.2 Auswirkungen auf die Testbarkeit

Die beschriebenen Querverbindungen der Anwendungsfälle sind im Programmcode über statische Methodenaufrufe fest verdrahtet. Nachstehend ein Ausschnitt aus dem Anwendungscode, der beispielsweise zur Entstehung der Abhängigkeit 9.2.3 führt:

```
abstract public class SVAendernErfassenAA extends SeminarisFrame {
...
   public void seminartypZuordnen() {
      boolean go = true;
      ...
      if (go) {
         SeminartypAuswaehlenAA seminartypAuswaehlenAA =
                                SeminartypAuswaehlenAA.erzeuge(this);
         String stSchluessel = seminartypAuswaehlenAA.oeffnenAuswahl();
      ...
```

Man erkennt, dass die Objekterzeugung mit Hilfe der statischen `erzeuge`-Methode der Klasse `SeminartypAuswaehlenAA` erfolgt. Dies führt an dieser Stelle wiederum zu den bereits mehrfach beschriebenen Problemen bei der Durchführung von *Unit*-Tests. Ohne Änderung der Implementierung der zu testenden Klasse kann ein Einsatz von Stellvertreterklassen nicht erfolgen,

- so dass der *Unit*-Test der abhängigen Klasse `SVAendernErfassenAA` erst durchgeführt werden kann, sobald die Server-Klasse `SeminartypAuswaehlenAA` implementiert ist.
- Tritt ein Fehler auf, kann sich die Fehlersuche im allgemeinen nicht mehr auf die zu testende Klasse `SVAendernErfassenAA` beschränken.
- Je größer die Anzahl der dadurch am Test beteiligten Klassen ist, desto höher sind die Zeiten für die Durchführung und den Ablauf der Testfälle.

11 Entfernung testkritischer Abhängigkeiten durch Refaktorisierung

11.1 Einleitung

Nachfolgend werden für einen Großteil der Entwurfsfehler, die in den Kapiteln 9 *Testbarkeitsanalyse ausgewählter Abhängigkeiten* und 10 *Entwurfsprobleme und Testbarkeit* als Ursachen für Testprobleme identifiziert wurden, konkrete Strategien zur alternativen Realisierung der jeweiligen Aufgaben beschrieben.

Auf Grundlage dieser Strategien erfolgt anschließend eine Refaktorisierung von SeminarIS. Während und am Ende der jeweiligen Refaktorisierungsphasen wird versucht, durch Beobachtung der Testbarkeitsmetriken Aussagen über die Auswirkungen der Umbauschritte und Hinweise auf weitere Probleme zu erhalten.

Um abschließend eine Einschätzung über den für die Refaktorisierung notwendigen Aufwand geben zu können, wird versucht, den Zeitaufwand für die jeweiligen Refaktorisierungsschritte festzuhalten.

11.2 Realisierung einer strikten Drei-Schichten-Architektur

11.2.1 Ausgangssituation

Obwohl als Vorgabe die Entwicklung einer strikten Drei-Schicht-Architektur galt, wurde aus drei Datenhaltungsklassen auf die Kontrollschicht zugegriffen (siehe Abschnitt 10.3). Hierbei handelt es sich zum einen um den Zugriff der Domänenklasse `Dozentvereinbarung` in Fassadenmanier auf die Kontrollklasse `DVKostenBerechnenK` (siehe auch Abschnitt 10.7). Bei den übrigen Zugriffen handelt es sich Versuche, in der Datenhaltung die Datenbankinstanz von der Hauptkontrollklasse zu erfragen. Ferner erfolgt aus fünf Kontrollklassen der Zugriff auf die Schnittstellenschicht, um von der Hauptschnittstellenklasse zur Ausgabe von Meldungen die Instanz des Hauptanwendungsfensters zu erhalten.

11.2.2 Durchführung des Refactoring

Schritt 1:
Die obsolete Methode `getKosten(...)` kann sowohl aus der Domänenklasse `Dozentenvereinbarung` als auch aus seinem Interface `IDozentenvereinbarung` entfernt werden. Prinzipiell könnte damit die komplette Klasse `DVKostenBerechnenK` entfernt werden und ebenfalls die Klasse `Zeitpunkt`, die im selben Source-File definiert ist. Dies führt aber dazu, dass das Projekt nicht mehr kompilierbar ist. In der Klasse `DozentenvereinbarungAuswaehlenAA` existiert Sourcecode für eine Schaltfläche zur Berechnung der Kosten eines Dozenten. Diese Schaltfläche ist aktuell jedoch nicht aktiviert, wie auch kein Anwendungsfall überhaupt eine entsprechende Funktionalität vorsieht. Dies erlaubt, die entsprechenden Passagen ebenfalls zu entfernen und in der Folge die Klassen `DVKostenBerechnenK` und `Zeitpunkt` aus dem System zu nehmen.

Schritt 2:
Der Zugriff auf die Datenbank kann bei den Klassen `Dozentvereinbarung` und `DozentvereinbarungTripel` über die von der Klasse ErweitertPersistent geerbte Methode `getDb()` erfolgen. In der Klasse `DozentvereinbarungAnfragen` kann für die

Refaktorisierung der Zugriff ohne Probleme auch direkt auf die einzige Instanz der Klasse `SeminarisDatenbank` erfolgen. Der Zugriff auf die Datenbank muss sicherlich in einer der folgenden Refaktorisierungsphasen von Grund auf neu gefasst werden.

Schritt 3:
Abweichend von der Implementierung aller übrigen Anwendungsfälle wurde in den Anwendungsfällen zur Dozentenvereinbarung in der Kontrollschicht die Ausgabe von Hinweisen und Meldungen an den Anwender realisiert. Überhaupt wurde in diesen Anwendungsfällen von den in der Entwurfsphase aufgestellten Design-Entscheidungen praktisch völlig abgewichen. Dies macht das Refactoring sehr aufwendig, da es durch das völlig andere Konzept stellenweise nicht möglich ist, die tatsächliche Implementierung an den geplanten Entwurf anzupassen. Hier handelt es sich letztlich nicht um einen versehentlichen Zugriff auf die Schnittstellenschicht, sondern um eine bewusste Vorgehensweise.

Für die Refaktorisierung fallen dabei folgende Aufwände an:

- **Schritt 1:** 5 Minuten
- **Schritt 2:** 2 Minuten
- **Schritt 3:** 160 Minuten

11.2.3 Ergebnisse

Man erhält folgende Veränderung der Projektmetriken:

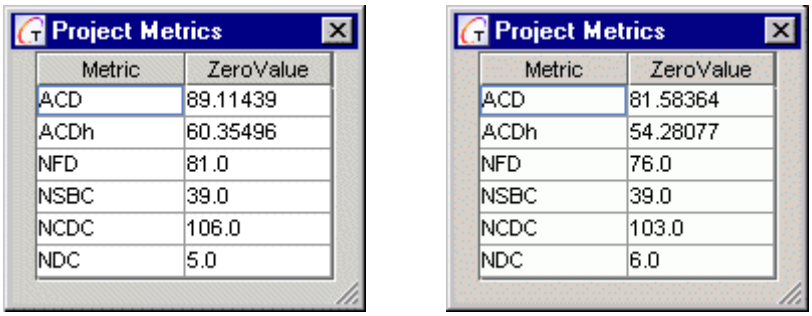


Abbildung 109: Projektmetriken vor und nach Entfernen von DVKostenBerechnenK

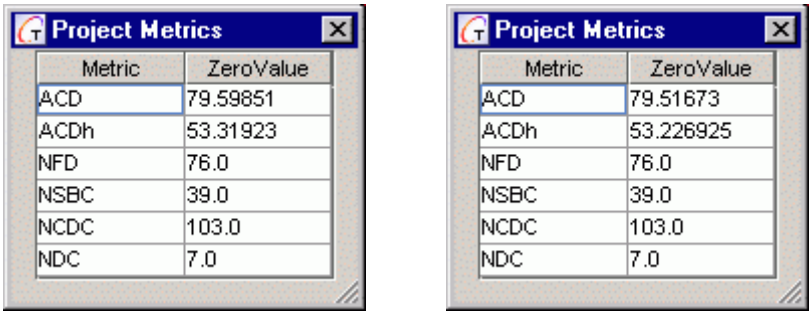


Abbildung 110: Projektmetriken nach den weiteren Schritten zur strikten Drei-Schicht-Architektur

Während der erste Refaktorisierungsschritt die deutlichste Veränderung brachte, führte die Entfernung der restlichen Abhängigkeiten aus der Datenhaltung in die Kontrollschicht nur mehr zu einer kleinen Veränderung der Testbarkeitsmetriken. Das mit dem meisten Aufwand verbundene Beseitigen der

Abhängigkeiten aus der Kontrollschicht in die Schnittstellenschicht ergab hingegen keine nennenswerte Veränderung.

Die Anzahl der Klassen, von denen eine Klasse im Durchschnitt direkt oder indirekt abhängig ist, ist um 10,8 Prozent gesunken, beschränkt auf fest verdrahtete Abhängigkeiten um 11,8 Prozent. Der große Abhängigkeitszyklus in der Datenhaltung wurde in drei kleinere voneinander unabhängige Zyklen aufgespalten, die Anzahl der zyklischen abhängigen Komponenten hat sich dabei nur unwesentlich verringert.

Ein Blick auf die Abhängigkeitsmetriken zeigt die erwarteten Veränderungen:

Abhängige Klasse	Funktionalität bereitstellende Klasse	dep_comp	is_feedb	is_intpk	rACD	rACDh	sub_hw	subedges	Typ der Abhängigkeit
SeminarisK	SeminarisDatenbank	0	0	1	5,9	11,6	1	5	static access
SVAendernErfassenAA	SeminartypAuswaehlenAA	0	0	1	2,7	4,1	1	2	create and access
DruckDokumentDK	SeminarisK	0	0	1	2,2	2,7	4	4	static access
SeminartypAendernErfassenAA	DozentenvereinbarungAuswaehlenAA	0	0	1	2,1	3,0	1	3	create and access
SVAendernErfassenAA	LeitungsauftragAuswaehlenAA	1	0	1	2,0	3,4	1	3	create and access
LeitungsauftragAendernErfassenAA	SVAuswaehlenAA	1	1	1	1,9	2,9	1	2	create and access
SeminarbelegungAA	SVAuswaehlenAA	0	0	1	1,8	2,8	2	3	create and access
ErweitertPersistent	SeminarisDatenbank	1	0	1	1,3	3,7	1	1	static access
TeilnahmeFirmenSVRelation	TeilnahmeFirmenSVPaar	1	0	0	1,1	1,4	4	4	create and access
SVSeminartypRelation	SVSeminartypPaar	1	0	0	1,1	1,4	5	5	create and access
SVAnsprechpartnerRelation	SVAnsprechpartnerPaar	1	0	0	1,1	1,4	4	4	create and access
AnstellungRelation	AnstellungPaar	1	0	0	1,1	1,4	5	5	create and access
BelegungRelation	BelegungTripel	1	0	0	1,1	1,4	6	6	create and access
BuchtRelation	BuchtPaar	1	0	0	1,1	1,4	5	5	create and access

Abbildung 111: Abhängigkeitsmetriken nach Entfernung aller Schichtenverletzungen

11.3 Kein externer Zugriff auf Kontrollschicht

11.3.1 Ausgangssituation

Durch Zugriff auf globale Variablen der Hauptkontrollklasse (`SeminarisK.Protocol` und `SeminarisK.Config`) aus den Paketen `ExterneSystemeP.ProtokollP`, `UtilP.MeldungP` und `ExterneSystemeP.DruckerP` entstehen zyklische Abhängigkeiten zwischen den beteiligten Paketen. Durch die Art der Implementierung von `SeminarisK` entstehen ferner transitiv Abhängigkeiten zur Datenbank und von dort zur Datenhaltungsschicht.

11.3.2 Durchführung des Refactoring

Alle vorhandenen Abhängigkeiten können durch den direkten Zugriff auf die in den globalen Variablen gespeicherten Instanzen (Singletons) ersetzt werden.

Beispiel bisheriger Zugriff über `SeminarisK`:

```
name = SeminarisK.Config.getPropertyMitDefault(
                          "Printer.BigFont.Name", "Courier");
```

Direkter Zugriff auf Konfigurationsinstanz:

```
name = Konfiguration.getEinzigeInstanz().getPropertyMitDefault(
                                        "Printer.BigFont.Name", "Courier");
```

Dies kann jedoch nur ein Zwischenschritt im Rahmen der Verbesserung des Entwurfs sein. Nach einem weiteren Refaktorisierungsschritt (siehe Abschnitt 11.4) wird der Zugriff nicht mehr fest verdrahtet auf die Singleton-Instanzen erfolgen. Damit kann die wünschenswerte Konfigurierbarkeit der Anwendung (insbesondere für Testzwecke) erreicht werden.

Der gesamte Aufwand für die Refaktorisierung beträgt 10 Minuten

11.3.3 Ergebnisse

Man erhält folgende Veränderung der Projektmetriken:

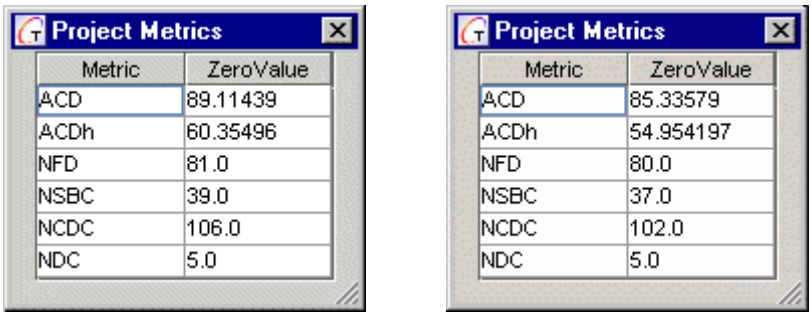


Abbildung 112: Projektmetriken nach Entfernen des externen Zugriffs auf Kontrollschicht

Die Anzahl der Klassen, von denen eine Klasse im Durchschnitt direkt oder indirekt abhängig ist, ist um 4,2 Prozent gesunken, beschränkt auf fest verdrahtete Abhängigkeiten um 8,9 Prozent. Die Anzahl der zyklischen abhängigen Komponenten hat sich geringfügig verringert.

Ein Blick auf die Abhängigkeitsmetriken zeigt, dass die betroffenen Abhängigkeiten weggefallen sind und die Werte der Abhängigkeit von `SeminarisK` zur Datenbank gesunken sind:

Abhängige Klasse	Funktionalität bereitstellende Klasse	dep_comp	is_feedb	is_intpk	rACD	rACDh	sub_hw	subedges	Typ der Abhängigkeit
Dozentenv ereinbarung	DVKostenBerechnenK	1	0	1	8,2	9,0	1	9	create and access
SVAendernErfassenAA	Seminarty pAusw aehlenAA	0	0	1	2,3	3,6	1	2	create and access
SeminarisK	SeminarisDatenbank	1	0	1	2,2	4,8	1	5	static access
SVAendernErfassenAA	LeitungsauftragAusw aehlenAA	1	0	1	1,9	3,3	1	3	create and access
Seminarty pAendernErfassenAA	DozentenvereinbarungAusw aehlenAA	0	0	1	1,7	2,5	1	3	create and access
LeitungsauftragAendernErfassenAA	SVAusw aehlenAA	1	1	1	1,7	2,8	1	2	create and access
SeminarbelegungAA	SVAusw aehlenAA	0	0	1	1,6	2,7	2	3	create and access
DVKostenBerechnenK	DozentenvereinbarungLoeschenK	1	1	0	1,6	1,8	1	1	static access
Erw eitertPersistent	SeminarisDatenbank	1	0	1	1,5	4,8	1	1	static access
SeminarisH	SachbearbeiterA	1	1	0	1,4	2,3	1	3	static access
SVKurzS	Seminarv eranstaltung	0	0	1	1,2	?	0	8	dep. to abstract class
MeldungsAnzeigeA	ExceptionNachrichtA	0	0	0	1,0	0,8	1	3	create and access
SeminarisK	LogProtokollDateiA	0	0	1	1,0	1,2	1	2	create and access
SeminarisK	DebugProtokollDateiA	0	0	1	1,0	1,2	1	2	create and access

Abbildung 113: Abhängigkeitsmetriken nach Entfernen des externen Zugriffs auf Kontrollschicht

11.4 Kein externer Zugriff auf Datenhaltung

11.4.1 Ausgangssituation

Die Verwaltung von eindeutigen Domänenbezeichnern erfolgte durch Ansiedlung entsprechender Logik in der Hauptkontrollklasse `SeminarisK` und der Datenbankklasse `SeminarisDatenbank`. Dies führte zum Zugriff auf acht Datenhaltungsklassen aus dem Basismodul `ExterneSystemeP.PersistenzP`. In Verbindung mit der fest verdrahteten Abhängigkeit zwischen `SeminarisK` und `SeminarisDatenbank` führte dies zu gravierenden Problemen (siehe Abschnitt 10.5).

11.4.2 Durchführung des Refactoring

Der vorliegende Entwurfsfehler wird beseitigt durch Verlagerung der `SeminarisDatenbank`-Methoden `getHoechste...nummer()` sowie der `SeminarisK`-Methoden `getNext...nummer()` in die Datenhaltungsschicht.

Im Einzelnen erfolgt dies durch Verschmelzung der Funktionalität der beiden jeweils zusammengehörenden `getHoechste...()` - und `getNext...()`-Methoden und deren Verlagerung in die Klassen

- `GeschaeftspartnerOrdner` (Verwaltung der Kundennummer als eindeutiges Attribut eines Geschäftspartners)
- `SeminarveranstaltungOrdner` (Verwaltung der Rechnungsnummer als eindeutiges Attribut einer Firmenseminarveranstaltung)
- `BelegungRelation` (Verwaltung der Rechnungsnummer als eindeutiges Attribut einer Belegung)
- `LeitungsauftragRelation` (Verwaltung der Leitungsauftragsnummer als eindeutiges Attribut eines Leitungsauftrags).

Dies führt unter anderem dazu, dass alle SQL-Abfragen auf die Datenbank ausschließlich in den Ordner- und Relationenklassen angesiedelt sind.

Die Verlagerung der Methoden muss abschließend in der Kontrollschicht nachvollzogen werden. Hierzu sind Anpassungen des Methodenaufrufs in fünf Kontrollklassen notwendig.

Für die Refaktorisierung fallen folgende Aufwände an:

- Leitungsauftrag: 15 Minuten
- Geschäftspartner: 10 Minuten
- Belegung: 10 Minuten
- Firmenseminarveranstaltung: 15 Minuten

Im jeweiligen Aufwand ist bereits die Anpassung des Zugriffs aus der Kontrollschicht enthalten.

11.4.3 Ergebnisse

Man erhält folgende Veränderung der Projektmetriken:

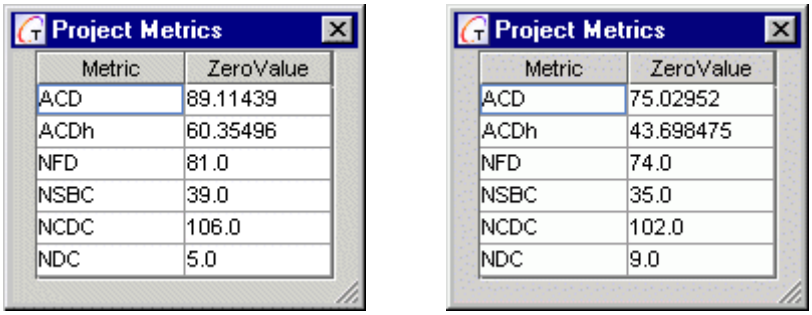


Abbildung 114: Projektmetriken vor und nach Entfernung der Abhängigkeit von der Datenbank zur Datenhaltung

Die Anzahl der Klassen, von denen eine Klasse im Durchschnitt direkt oder indirekt abhängig ist, ist um 15,4 Prozent gesunken, beschränkt auf fest verdrahtete Abhängigkeiten sogar um 27,1 Prozent. Der große Abhängigkeitszyklus in der Datenhaltung wurde in fünf kleinere voneinander unabhängige Zyklen aufgespalten, die Anzahl der zyklischen abhängigen Komponenten hat sich dabei nur unwesentlich verringert.

Ein Blick auf die Abhängigkeits- und Komponentenmetriken zeigt insbesondere, dass die rACD-Werte sowohl für die (nach wie vor vorhandene) Abhängigkeit von `SeminarisK` nach `SeminarisDatenbank` (Verringerung rACD von 6,05 Prozent auf 0,19 Prozent) als auch für beide Klassen deutlich gesunken sind:

Klasse	vor Refactoring		nach Refactoring	
	r_ACD_in	rACD_out	r_ACD_in	rACD_out
SeminarisK	8,87	8,27	3,70	2,71
SeminarisDatenbank	17,79	16,89	2,68	1,19

Abbildung 115: Komponentenmetriken vor und nach Entfernung der Abhängigkeit von der Datenbank zur Datenhaltung

Von dieser Refaktorisierung betroffen sind auch alle anderen testkritischen Abhängigkeiten, die von SeminarisDatenbank abhängen. Beispielsweise sinken die Werte der Metriken rACD und rACDh für die Abhängigkeit von `ErweitertPersistent` nach `SeminarisDatenbank` (siehe Abschnitt 9.2.11) von 1,4 und 4,4 auf 0,1 und 0,6.

11.5 Veränderter Zugriff auf globale Basiskomponenten

11.5.1 Ausgangssituation

Die aktuelle Form des Zugriffs auf globale Basiskomponenten führt

- zu einer Erhöhung der Abhängigkeiten im System durch die zentrale Bündelung in der Klasse `SeminarisK`,
- infolge von schichtenverletzenden Zugriffen zu zyklischen Abhängigkeiten innerhalb der Anwendung,

- führt aufgrund der fest verdrahteten Abhängigkeiten bei der Objekterzeugung zu einer mangelnden Austauschbarkeit der Basiskomponenten in der Testumgebung und damit zu einer maßgeblichen Beeinträchtigung des Testprozesses.

11.5.2 Durchführung des Refactoring

Für die Realisierung des Zugriffs auf globale Basiskomponenten eines Systems stehen mehrere Möglichkeiten zur Verfügung. Nachfolgend werden zunächst einige dieser Konzepte vorgestellt und auf ihre Vor- und Nachteile hin untersucht. Aus Sicht der Testbarkeit ist bei dieser Untersuchung die Frage von zentraler Bedeutung, wie ein Austausch der Basiskomponenten in der Testumgebung erreicht werden kann.

11.5.2.1 Direkter Zugriff auf die Basiskomponenten

Die Realisierung der Basiskomponenten erfolgt zumeist nach dem Entwurfsmuster Singleton [Gamm94]. Über die statische Methode zur Erlangung der einzigen Instanz (z.B. `getInstance()`) ist es möglich, an jedem Punkt der Anwendung global auf die Komponente zuzugreifen.

Mit dem direkten Zugriff wäre zwar das Problem der Bündelung und Abhängigkeit aller Basiskomponenten beseitigt, jedoch bleibt der fest verdrahtete Zugriff auf die einzige Instanz und die fehlende Austauschbarkeit der Komponente in der Testumgebung.

Aus Sicht der Testbarkeit bringt diese Lösung deshalb keinen Vorteil.

11.5.2.2 Übergabe der Basiskomponenten bei der Konstruktion

Die Berücksichtigung von Testbarkeitsbelangen besitzt einen sehr hohen Stellenwert innerhalb der testgetriebenen Entwicklung (Test-First Design, ein wesentliches Element von Xtreme Programming). Üblicherweise wird dort die Bereitstellung von Basiskomponenten durch Übergabe der Komponente als Parameter an den Konstruktor einer Klasse gelöst (siehe beispielsweise [Link01]). Durch die Verwendung von Interfacetypen als Typen der Übergabeparameter wird die Abhängigkeit der nutzenden Klassen von einer konkreten Instanz vermieden.

Beispiel für die Übergabe der Datenbank in der Datenhaltung:

```
public class EntityClass {
   public EntityClass( DatabaseInterface db, …) {
   …
   }
}
```

Jede Klasse, die Zugriff auf eine Basiskomponente (z.B. die Datenbank) benötigt, erhält wie in obigem Beispiel eine den Interfacetyp implementierende Ressource bei der Objekterzeugung übergeben. Bei der Erstellung von Testfällen hat dies den großen Vorteil, dass die Verwendung eines für Testzwecke implementierten Teststellvertreters (vom Interfacetyp) auf einfache Art durch Parameterübergabe im Rahmen der Initialisierung des Testfalls erfolgen kann.

Es liegt in der Natur der Sache, dass dieses Parameterkonzept in der Literatur zumeist an kleinen, einfachen und einprägsamen Beispielen veranschaulicht wird. Daher bleibt die Frage unbeantwortet, wie sinnvoll und zweckmäßig dieses Konzept für den Entwurf eines größeren Projekts sein kann. Am Beispiel des Datenbankzugriffs in *SeminarIS* sollen die Konsequenzen aus dem beschriebenen Vorgehen illustriert werden:

- In *SeminarIS* erfolgt augenblicklich der Zugriff auf die Datenbank sowohl in den Kontrollklassen zur Transaktionssteuerung als auch in der Datenhaltung zur Abfrage oder Manipulation der Datenbasis. Die Erzeugung der Kontrollklassen erfolgt in den zum jeweiligen Anwendungsfall gehörenden Schnittstellenklassen. Geht man davon

aus, dass die Instanz der Datenbank bei der Initialisierung der Anwendung (z.B. in der Hauptschnittstellenklasse) erzeugt wird, würde dies bedeuten, dass die Hauptschnittstellenklasse bereits jeder Schnittstellenklasse bei deren Erzeugung die Datenbankinstanz als Parameter übergeben muss. Die Schnittstellenklassen wiederum reichen die Datenbankinstanz an die Kontrollklassen bei deren Instantiierung weiter. Die Kontrollklassen schließlich reichen die Instanz dann beim Zugriff auf die Datenhaltung an die Entitätsklassen durch. Verfährt man analog auch für andere globale Basiskomponenten des System (Protokollinstanz, Druckerinstanz, Konfigurationsinstanz, ...) so kann dies in manchen Klassen zu einer wahren Inflation von Konstruktorparametern führen.

- Wird die Erzeugung der globalen Ressourcen bei der Initialisierung der Anwendung vorgenommen, so erfolgt dies unter Umständen lange vor der erstmaligen Nutzung oder möglicherweise sogar ohne eine spätere Verwendung der Basiskomponente.
- Man könnte aufgrund der vorstehenden Anmerkungen versucht sein, die globalen Basiskomponenten erst bei konkretem Bedarf in der Anwendung zu erzeugen. So könnte sich beispielsweise eine Kontrollklasse die Datenbankinstanz erst unmittelbar vor dem Aufruf einer Datenhaltungsklasse (mit Datenbankzugriff) besorgen. Dann landet man aber wieder beim Ausgangsproblem, wie ein globaler Zugriff auf die Datenbankinstanz an dieser Stelle am günstigsten erfolgen sollte.

Unter Abwägung aller Für und Wider scheint das beschriebene Konzept der Parameterübergabe in Verbindung mit der Definition des Parameters als Interfacetyp dennoch ein gut geeignetes Instrument für den Umgang mit Basiskomponenten zu sein. Aus den Erfahrungen mit *SeminarIS* scheint ein Nachteil lediglich zu sein, dass das Konzept an beliebigen Stellen der Anwendung durch direkten Zugriff auf die Basiskomponenten umgangen werden kann.

Für die Refaktorisierung von *SeminarIS* kommt es auch deshalb nicht zur Anwendung, weil die dort vorhandene Struktur zu einem sehr hohen Refaktorisierungsaufwand bei der Umsetzung dieses Musters führen würde.

11.5.2.3 Verwendung des Entwurfsmusters Factory zur Objekterzeugung

Ein verbreiteter Ansatz um Unabhängigkeit von der Objekterzeugung zu erreichen, ist die Verwendung des Entwurfsmusters *AbstractFactory* (siehe [Gamm94]). Für den Zugriff auf die Datenbankinstanz könnte die Realisierung beispielsweise wie folgt aussehen:

Zugriff auf die Datenbank in der Anwendung:

```
public class Firma {
   private Datenbank db;
   public Firma( AbstrakteDatenbankFabrik df){
      db = df.getDatenbank();
   }
   …
}
```

Implementierung der abstrakten Fabrikklasse nach dem Entwurfsmuster *AbstractFactory*:

```
abstract class AbstrakteDatenbankFabrik {
   abstract Datenbank getDatenbank();
}
```

Konkrete Bereitstellung der Datenbank in der Anwendungsumgebung:

```
class LaufzeitDatenbankFabrik {
   public Datenbank getDatenbank() {
      return SeminarisDatenbank.getEinzigeInstanz();
}
```

Konkrete Bereitstellung der Datenbank in der Testumgebung:

```
class TestumgebungDatenbankFabrik {
   public Datenbank getDatenbank() {
      return new TestDatenbank();
}
```

Mit diesem Entwurfsmuster wird allerdings nur die Unabhängigkeit von der Objekterzeugung erreicht. Die Frage des Zugriffs auf die globale Basiskomponente bleibt unbeantwortet. In unserem Beispiel findet erneut das Konzept der Parameterübergabe Anwendung. Insoweit ergibt sich kein Fortschritt bei der Frage des Zugriffs.

Der direkte Zugriff auf eine der Fabriken verbietet sich, da damit erneut eine statische Abhängigkeit erreicht würde. Alternativ wäre denkbar, die im aktuellen Kontext gültige Fabrik von einer Systemklasse abzufragen (z.B. `MySystem.getDatenbankFabrik()`). Dies würde jedoch eine erneute Bündelung der Basiskomponenten bedeuten, was eine verstärkte Kopplung des Systems nach sich ziehen würde. Zudem kann weiterhin unter Umgehung der Fabrik die Objekterzeugung direkt erfolgen.

Dass das Entwurfsmuster hier keine Hilfestellung ist, liegt vermutlich daran, dass es hier eine gewisse Zweckentfremdung erfährt. Die originäre Definition und Aufgabenstellung des Patterns war die dynamische und variable Objekterzeugung zur Laufzeit des Programms (siehe **[**Gamm94**]** und **[**Coop98**]**). Im hier diskutierten Fall handelt es sich aber nicht um dynamische Objekterzeugung, da für die Anwendung stets die Erzeugung derselben Objektinstanz vorgesehen ist.

11.5.2.4 Zugriff über Registry-Mechanismus

Ein verbreiteter Ansatz, der auf die globale Bereitstellung von Systemressourcen bei gleichzeitiger Unabhängigkeit von der Objekterzeugung zielt, ist deren Bereitstellung über einen *Registry*-Mechanismus oder Namensdienst (siehe **[**Coop98**]** und **[**Wulff02**]**).

Hierbei wird beispielsweise jede *Singleton*-Instanz (als die häufigste Implementierung von Basiskomponenten) nach Ihrer Erzeugung bei einer *Registry*(-Klasse) angemeldet. Jede Klasse, die Zugriff auf ein *Singleton* benötigt, bekommt in der Folge von der *Registry* über Angabe eines textuellen Bezeichners (z.B. den Namen der Klasse) die erzeugte Instanz des *Singletons* geliefert.

Die Unabhängigkeit von der Objekterzeugung wird bei dieser Variante nur auf Kosten von reduzierter Typsicherheit erreicht: Da die *Singleton*-Instanzen in der *Registry* üblicherweise als Instanzen vom Typ `Object` in beispielsweise einer `Hashtable` gehalten werden, sind die Möglichkeiten der Typüberprüfung eingeschränkt (ähnlich wie bei einer grundsätzlich natürlich auch denkbaren *Reflection*-Lösung).

Auch hier erfolgt die Erzeugung der Singleton-Instanzen üblicherweise bei der Initialisierung der Anwendung (z.B. in der Hauptkontrollklasse) und damit vor dem eigentlichen Zugriffszeitpunkt. Zudem ist weiterhin ein versehentlicher direkter Zugriff auf die Singleton-Instanzen innerhalb der Anwendung möglich.

Zuletzt stellt sich auch hier wieder die Ausgangsfrage: Wie erfolgt der Zugriff auf die *Registry*? Allerdings ist hier die Realisierung als *Singleton* und der statische Zugriff auf die einzige Instanz ohne Nachteile hinsichtlich der dynamischen Konfigurierbarkeit der Anwendung möglich.

Nimmt man also die fehlende Typsicherheit in Kauf, so handelt es sich um ein durchaus probates Mittel für den Zugriff auf globale Komponenten. Nicht zuletzt deshalb, weil im Rahmen der verteilten und komponentenorientierten Programmierung komplette Anwendungen auf Konzepten beruhen, die diesen Mechanismus verwenden. Als Stichpunkte seien nur CORBA und wiederum J2EE genannt.

11.5.2.5 Auf Seminaris zugeschnittenes Alternativkonzept

Für die Refaktorisierung von *SeminarIS* findet unter Beachtung der vorstehenden Überlegungen ein Alternativkonzept Verwendung. Es basiert auf der Entfernung der globalen Zugriffsmöglichkeit aus den *Singleton*-Klassen und auf der Einführung einer konfigurierbaren Zugriffsklasse für jede *Singleton*-Instanz. Diese Variante garantiert folgende Punkte:

- Dynamische Konfigurierbarkeit der Basiskomponenten in der Testumgebung,
- Erzeugung der Ressourcen bei Bedarf,
- keine zusätzlichen Abhängigkeiten durch Bündelung des Zugriffs auf die Basiskomponenten an einer Stelle des Systems,
- keine Reduzierung der Typsicherheit,
- kein versehentliches oder absichtliches Umgehen des Konzeptes möglich,
- relativ geringer Implementierungsaufwand.

Die konkrete Realisierung des Konzept wird nachstehend am Beispiel des Zugriffs auf die Datenbank von *SeminarIS* erläutert:

Jede Basiskomponente erhält innerhalb ihres Pakets eine `Provider`-Klasse zur Seite gestellt, die über eine statische `get...()`-Methode den globalen Zugriff auf die Basiskomponente sicherstellt. Diese `get...()`-Methode liefert als Rückgabetyp einen noch zu definierenden Interfacetyp:

```
package SeminarisP.ExterneSystemeP.PersistenzP;

public class SeminarisDatenbankProvider {
    private static ISeminarisDatenbank seminarisDatenbank =
                                    SeminarisDatenbank.getEinzigeInstanz();

    public static ISeminarisDatenbank getDatenbank() {
        return seminarisDatenbank;
    }

    public static void setDatenbank( ISeminarisDatenbank aktuelleDatenbank){
        seminarisDatenbank = aktuelleDatenbank;
    }
}
```

Ferner stellt die Providerklasse eine `set...()`-Methode zur Verfügung, über die beispielsweise in der Testumgebung die dynamische Konfiguration der Datenbank erfolgen kann.

Diskussionwürdig ist sicher die vorhandene Initialisierung der Providervariable mit der Laufzeitinstanz von `SeminarisDatenbank`. Gegen diese Initialisierung spricht die dadurch transitiv entstehende syntaktische Abhängigkeit von dieser Klasse. Aus Sicht der Testbarkeit birgt dies jedoch nur den Nachteil, dass für den Fall der (noch) fehlenden Implementierung der Klasse bei der Testfallerstellung eine Dummy-Klasse erstellt werden muss. Dem stehen aber zwei Vorteile gegenüber: Die Erzeugung der Instanzen erfolgt nicht schon beim Starten des Systems (z.B. in der Hauptschnittstellenklasse) und der Zugriff auf die statische Erzeugermethode `getEinzigeInstanz()` der Singletonklasse SeminarisDatenbank kann paketlokal gesetzt werden, da kein Zugriff auf die Instanz von außerhalb des Pakets erfolgen muss:

```
public class SeminarisDatenbank extends Datenbank
                                implements ISeminarisDatenbank
{
    /** Einzige Instanz der Klasse (Entwurfsmuster Einzelstueck). */
    private static SeminarisDatenbank einzigeInstanz = null;

    /** Standard-Konstruktor soll nicht mehr genutzt werden. */
    protected SeminarisDatenbank( int dbms) {
     ....
    }

    /** Liefert die einzige Instanz der Seminaris-Datenbank. */
    synchronized static SeminarisDatenbank getEinzigeInstanz() {
        .....
    }
```

Durch den paketlokalen Zugriffsmodifikator kann verhindert werden, dass außerhalb des Paketes ein direkter Zugriff auf das Einzelstück der Datenbank erfolgt.

Für die Ersetzbarkeit der Datenbank in der Testumgebung ist ferner Voraussetzung, dass beim Zugriff auf die Providerklasse als Rückgabewert lediglich ein Interfacetyp zugesichert wird. So kann jede beliebige Implementierung dieses Interfaces als aktuell zu verwendende Datenbank gesetzt werden.

Während bei den übrigen SeminarIS-Basiskomponenten die Implementierung des Interfaces trivial ist, muss zur Verkapselung der externen Rahmenwerksklasse `persistenz.Datenbank` das Interface `ISeminarisDatenbank` zusätzlich als Adapter fungieren. Dies erfolgt unter Ausnutzung der in Java vorhandenen Möglichkeit, dass gleich lautende Methodensignaturen in Superklassen und implementierten Interfaces vorhanden sein dürfen:

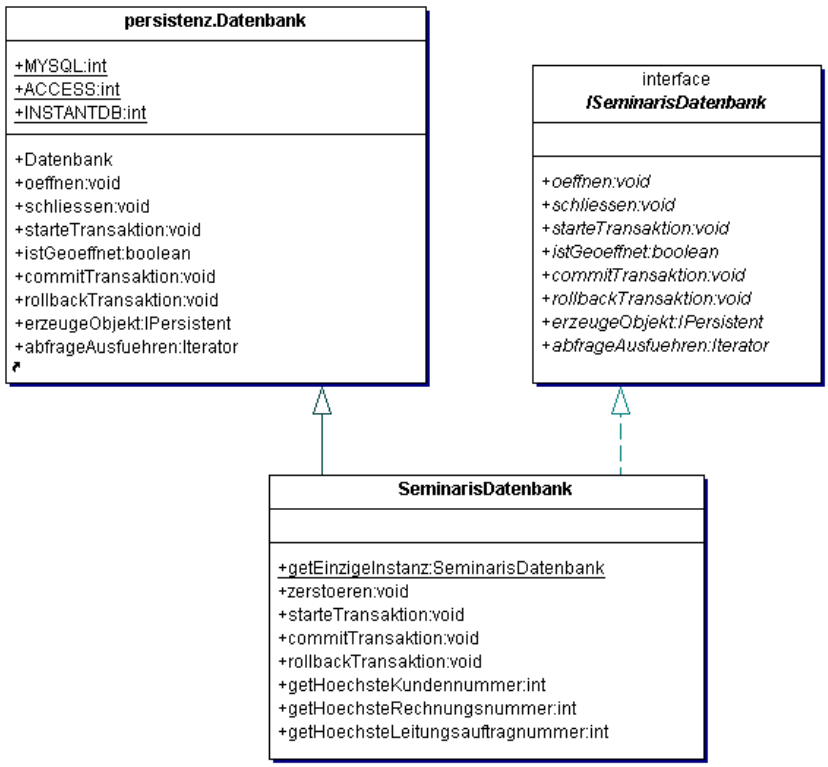


Abbildung 116: Abkopplung vom Persistenz-Rahmenwerk durch Interface ISeminarisDatenbank

Abschließend müssen im gesamten Projekt die Zugriffe auf die Datenbank angepasst werden.

Dieses Vorgehen führen wir analog auch für folgende Basiskomponenten durch:

- `Konfiguration`,
- `SynchronisationsManager`,
- `ProtokollManager`,
- `DebugProtokollDateiA`,
- `LogProtokollDateiA`.

Nach Abschluss dieser Refaktorisierung und der Verlagerung der Logik zur Verwaltung von eindeutigen Domänenattributen (siehe Abschnitt 11.4) aus der Klasse `SeminarisK` in die Datenhaltung kann die Klasse `SeminarisK` entfallen.

Der Gesamtaufwand für die Refaktorisierung beträgt 90 Minuten.

11.5.3 Ergebnisse

Die Refaktorisierung wirkt sich wie folgt auf die Projektmetriken aus:

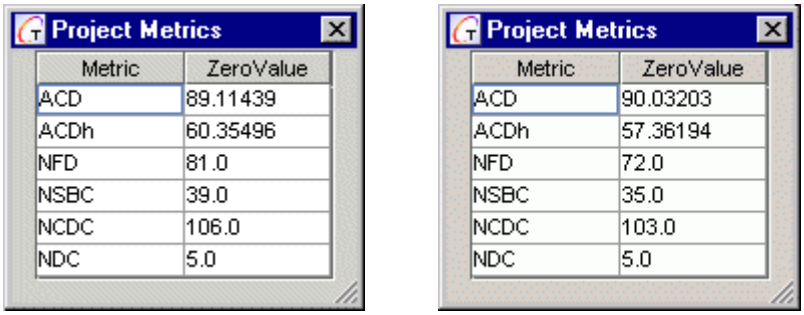


Abbildung 117: Projektmetriken vor und nach Änderung des Zugriffs auf Datenbank

Wie bereits erwartet ist der ACD-Wert nahezu unverändert geblieben, da durch die Initialisierung der Providerklassen die bestehenden Abhängigkeiten transitiv erhalten geblieben sind. Dennoch wurde eine Verbesserung der Testbarkeit erreicht.

11.6 Direkter Zugriff auf Ordner und Relationen

11.6.1 Ausgangssituation

Wie in Abschnitt 10.7 beschrieben, führt die Fassadenfunktion der Domänenklassen sowohl zu erhöhtem Implementierungsaufwand als auch in der Folge zu erhöhtem Testaufwand.

Die Testbarkeitsmetriken auf Klassenebene schlagen allerdings keinen Alarm, da die Abhängigkeiten breit gestreut auf viele Klassen verteilt sind. Aus dem Paketdiagramm (Abbildung 107: Zugriff auf Datenhaltung mit Fassadenmuster) wird aber sofort deutlich, dass es sich auf Paketebene um eine klassische Hub-Abhängigkeit handelt (`SeminarisP` ist der Hub). Insoweit lässt sich festhalten, dass eine Erweiterung der Testbarkeitsmetriken auf Paketebene nach wie vor absolut wünschenswert wäre.

Ein Indiz für das vermutete Ausschlagen der Metriken auch auf Paketebene liefert eine Übersicht der Abhängigkeiten mit den höchsten rACD-Metriken, die während des Refactoring erstellt wurde. Zum

Zeitpunkt der Errechnung der dort angezeigten Testbarkeitsmetriken war bereits sämtliche Fassaden- und darüber hinaus gehende Geschäftslogik aus den Entitätsklassen entfernt, mit einer Ausnahme: In der Klasse `Person` waren noch drei Methoden verblieben, die aufgrund umfassend vorhandener Logik nur mit erhöhtem Aufwand refaktoriert werden konnten:

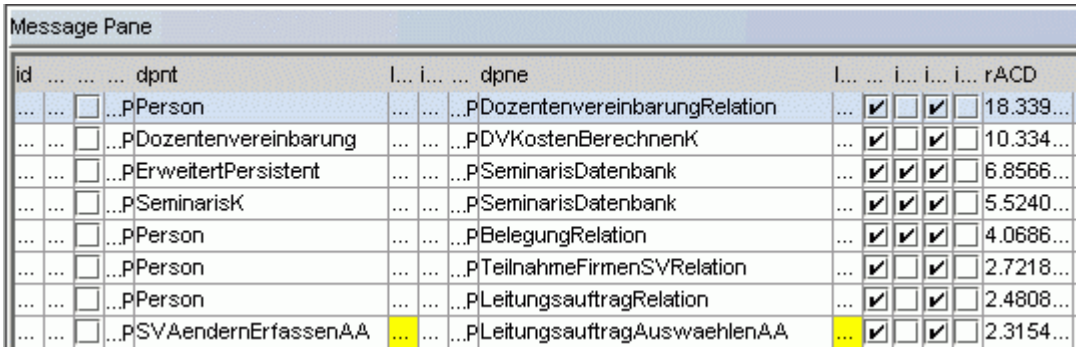

Message Pane
id ... ... dpnt l... i... ... dpne l... ... i... i... i... rACD
... ...P Person ... ...P DozentenvereinbarungRelation ... 18.339...
... ...P Dozentenvereinbarung ... ...P DVKostenBerechnenK ... 10.334...
... ...P ErweitertPersistent ... ...P SeminarisDatenbank ... 6.8566...
... ...P SeminarisK ... ...P SeminarisDatenbank ... 5.5240...
... ...P Person ... ...P BelegungRelation ... 4.0686...
... ...P Person ... ...P TeilnahmeFirmenSVRelation ... 2.7218...
... ...P Person ... ...P LeitungsauftragRelation ... 2.4808...
... ...P SVAendernErfassenAA ... ...P LeitungsauftragAuswaehlenAA ... 2.3154...

Abbildung 118: Momentaufnahme der Testbarkeitsmetriken während Datenhaltung-Refaktorisierung

Die in der Klasse `Person` zu diesem Zeitpunkt noch verbliebenen Abhängigkeiten zu Relationenklassen verfügen über deutlich erhöhte Metrikwerte.

11.6.2 Durchführung des Refactoring

Zur Refaktorisierung der Datenhaltungsschicht wurden sukzessive alle Domänen- und Assoziationsklassen auf das Vorhandensein von Abhängigkeiten zu Relationen- und Ordnerklassen untersucht. Das Vorgehen beim Entfernen der Abhängigkeiten unterschied sich nach dem Grad der Geschäftslogik, die in den zu refaktorierenden Fassadenmethoden vorhanden war.

Weitestgehend problemlos verlief die Entfernung derjenigen Methoden, die sich auf ein Durchreichen des Methodenaufrufs in die Ordner- oder Relationenklassen beschränkten. In diesen Fällen konnte in der Entitätsklasse die implementierte Methode und in der zugehörigen Interface-Klasse die Deklaration der Methode entfernt werden. Alle Zugriffe auf die Methode wurden ersetzt durch direkten Aufruf der Methoden in den Ordner- und Relationenklassen.

Der Aufwand für die Refaktorisierung betrug in etwa 6 ½ Stunden und ergibt sich aufgeschlüsselt nach Komponenten aus nachfolgender Übersicht:

Refaktorierte Klasse (jeweils einschließlich implementiertes Interface)	Aufwand in Minuten
Belegung	30
Firma	45
FirmenSeminarveranstaltung	50
Geschaeftspartner	15
Leitungsauftrag	20
OeffentlicheSeminarveranstaltung	30
Person	105
Seminartyp	35
Seminarveranstaltung	50
Dozentenvereinbarung	10
Gesamt	390

Abbildung 119: Aufwand für Entfernung von Methoden ohne Logik

Wesentlich aufwendiger gestaltete sich die Entfernung aller Methoden, bei denen der Aufruf der Methoden der Ordner- und Relationenklassen in zusätzliche Logik eingebettet waren. Nachteilig

wirkte sich an dieser Stelle aus, dass innerhalb der Entitätsklassen auch umfangreiche Geschäftslogik implementiert war, die sinnvollerweise in die Kontrollschicht gehört hätte. Bei der Refaktorisierung wurden demzufolge große Teile der vorhandenen Logik in die Kontrollschicht verlagert (in die Nähe der aufrufenden Methoden), der Rest der Logik wanderte in die Ordner- und Relationenklassen zu den aufgerufenen Methoden. Alle Aufrufe der Fassadenmethoden aus der Kontrollschicht konnten anschließend auch hier in direkte Aufrufe der eingebetteten Methoden der Ordner- und Relationenklassen umgewandelt werden.

Der Aufwand für diesen Teil der Umgestaltung betrug etwa 9 ½ Stunden. Eine Zuordnung des Aufwands auf einzelne Klassen ergibt sich aus folgender Übersicht:

Refaktorierte Klasse (jeweils einschließlich implementiertes Interface)	Aufwand in Minuten
Belegung	35
Firma	30
FirmenSeminarveranstaltung	10
Geschaeftspartner	40
Leitungsauftrag	10
OeffentlicheSeminarveranstaltung	35
Person	210
Seminartyp	30
Seminarveranstaltung	60
Dozentenvereinbarung	65
Gesamt	525

Abbildung 120: Aufwand für Entfernung von Methoden mit Logik

11.6.3 Ergebnisse

Nach der Entfernung sämtlicher Fassaden- und Geschäftslogik aus der Datenhaltung ergibt sich folgendes Bild für die Projektmetriken:

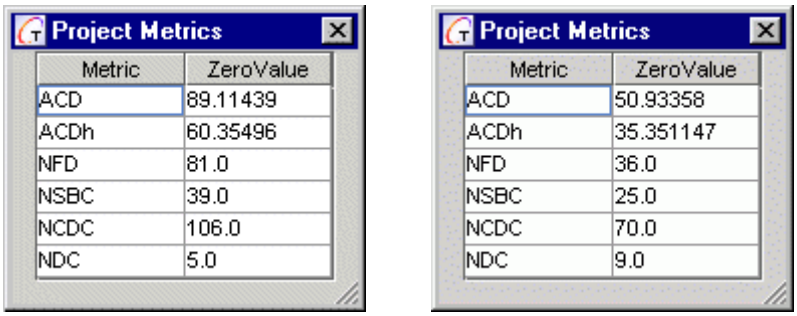


Abbildung 121: Projektmetriken vor und nach Entfernung der Fassaden- und Geschäftslogik aus Datenhaltung

Das Refactoring führt also zu einer Verringerung der Anzahl der Klassen, von denen eine Klasse im Durchschnitt direkt oder indirekt abhängig ist, um 42,6 Prozent. Bezogen auf ausschließlich fest verdrahtete Abhängigkeiten beträgt der Rückgang 41,0 Prozent. Die Anzahl der in Zyklen involvierten Komponenten nimmt um 34,0 Prozent ab. Die Erhöhung der Abhängigkeitszyklen von 5 auf 9 lässt sich als Auflösung eines allumfassenden Zyklus in der Datenhaltung in mehrere kleinere Zyklen interpretieren.

Durch das Refactoring ergibt sich auch bei den Abhängigkeiten mit dem höchsten rACD-Wert eine Verschiebung hin zu den Abhängigkeiten in der Schnittstellenschicht.

Message Pane

id	...	...	...	dpnt	l...	i...	...	dpne	l...	...	is...	i...	i...	rACD	rACDh
...	...	☐	...P	ErweitertPersistent	...	...	...P	SeminarisDatenbank	...	☑	☑	☑	☐	5.4263...	2.1917572
...	...	☐	...P	SVAendernErfassenAA	...	...	...P	LeitungsauftragAuswaehlenAA	...	☑	☐	☑	☐	3.5499...	6.7264175
...	...	☐	...P	SVAendernErfassenAA	...	...	...P	SeminartypAuswaehlenAA	...	☐	☐	☑	☐	3.3470...	4.988121
...	...	☐	...P	IDozentenvereinbarung	...	...	...P	Dozentenvereinbarung	...	☑	☑	☐	☐	2.8761...	NaN
...	...	☐	...P	SeminarisK	...	...	...P	SeminarisDatenbank	...	☐	☑	☑	☐	2.8689...	4.005615
...	...	☐	...P	LeitungsauftragAendernErfass...	...	...	...P	SVAuswaehlenAA	...	☑	☑	☑	☐	2.731285	4.2107544
...	...	☐	...P	SeminarbelegungAA	...	...	...P	SVAuswaehlenAA	...	☐	☐	☑	☐	2.5936...	4.2539444
...	...	☐	...P	SeminartypAendernErfassenAA	...	...	...P	DozentenvereinbarungAuswaehl...	...	☐	☐	☑	☐	2.564659	3.0878983
...	...	☐	...P	Dozentenvereinbarung	...	...	...P	DVKostenBerechnenK	...	☑	☐	☑	☐	2.4777...	1.4467773

Abbildung 122: Top-rACD-Abhängigkeitsmetriken nach Entfernung des Fassadenmusters

Eine andere Übersicht über die Veränderungen in der Datenhaltungsschicht erhält man, indem man die Anzahl der Programmzeilen und die Anzahl der Methoden in den Datenhaltungsklassen vor und nach der Refaktorisierung gegenüberstellt. Zum besseren Überblick wurden alle Komponenten ausgeblendet, bei denen sich keine Veränderung ergab:

Paket/Klasse/Interface	Vor Refactoring		Nach Refactoring	
	Lines Of Code	Number Of Operations	Lines Of Code	Number Of Operations
SeminarisP.DatenhaltungP	5441	964	4255	634
OrdnerP	243	15	425	27
GeschaeftspartnerOrdner	69	5	193	13
SeminartypOrdner	49	3	81	5
SeminarveranstaltungOrdner	125	7	151	9
SeminarisP	3878	833	2156	469
Belegung	415	41	334	32
Dozentenvereinbarung	408	47	203	31
Firma	247	34	119	15
FirmenSeminarveranstaltung	178	35	133	21
Geschaeftspartner	353	49	248	38
IBelegung	42	34	36	28
IFirma	35	28	19	12
IFirmenSeminarveranstaltung	30	25	17	12
IGeschaeftspartner	49	41	46	38
ILeitungsauftrag	24	17	20	13
IOeffentlicheSeminarveranstaltung	22	17	12	7
IPerson	94	84	29	19
ISeminartyp	45	37	24	16
ISeminarveranstaltung	77	71	61	55
Leitungsauftrag	124	22	87	15
OeffentlicheSeminarveranstaltung	191	28	117	16
Person	716	95	168	23
Seminartyp	326	43	152	19
Seminarveranstaltung	502	85	331	59
TransformationenP	1320	116	1674	138
AnstellungRelation	189	12	196	12
BelegungRelation	368	23	448	30
DozentenvereinbarungRelation	148	23	274	26
DozentenvereinbarungTripel	92	15	94	15
LeitungsauftragRelation	196	20	269	27
RechnungsempfaengerRelation	210	12	235	13
TeilnahmeFirmenSVRelation	117	11	158	15

Abbildung 123: Veränderung Lines Of Code (LOC)/Number of Operations (NOO) in der Datenhaltung

Man erkennt, dass die Anzahl der implementierten Methoden in der Datenhaltungsschicht um etwa ein Drittel (von 964 auf 634) abgenommen hat. Auch dies lässt auf eine deutliche Reduzierung des Testaufwandes schließen (vergleiche auch Abschnitt 10.7).

11.7 Einführung einer Schnittstellenklassen-Factory

11.7.1 Ausgangssituation

Wie in Abschnitt 10.8 beschrieben, führt die Abhängigkeit von Anwendungsfällen über *exceptional* oder *alternative flows* durch die direkte Abbildung dieser Abhängigkeiten beim Entwurf der Schnittstellenschicht zu einer starken Abhängigkeit der zu den Anwendungsfällen entworfenen Schnittstellenklassen. Zur Auflösung dieser Abhängigkeiten erfolgt die Implementierung einer Factory-Klasse, die für die Erzeugung der Schnittstellenklassen zuständig ist. Die Implementierung bleibt darauf beschränkt, nur die Auswahlklassen (`SeminartypAuswaehlenAA`, `LeitungsauftragAuswaehlenAA`, etc.) über den Factory-Mechanismus zu verwalten, da zunächst nur deren Erzeugung bei den Testbarkeitsmetriken ausschlägt. Die Verwaltung der übrigen

Schnittstellenklassen kann jederzeit nachgezogen werden. Insbesondere dann, wenn nach Einführung des Factory-Mechanismus für die Auswahlklassen die Testbarkeitsmetriken für die übrigen Schnittstellenklassen höhere Werte liefern würden.

11.7.2 Durchführung des Refactoring

Eine detaillierte Beschreibung der verschiedenen Nuancen des *Factory-Pattern* findet man in **[**Gamm94**]** und **[**Coop98**]**. Um völlige Unabhängigkeit vom Erzeugungsmechanismus zu erreichen, verwenden wir das Entwurfsmuster abstrakte Fabrik (*Abstract Factory*).

Wir beginnen zunächst mit der Definition des Interfaces für die Schnittstellenfabrik:

```
public interface IAAKlassenFabrik {
   AuswaehlenListeAA getFirmaAuswaehlenAA( SeminarisFrame parent);
   IPersonAuswaehlenAA getPersonAuswaehlenAA( SeminarisFrame parent);
   AuswaehlenListeAA getGeschaeftspartnerAuswaehlenAA(
                                          SeminarisFrame parent);
   AuswaehlenListeAA getBelegungAuswaehlenAA( SeminarisFrame parent);
   IDozentenvereinbarungAuswaehlenAA getDozentenvereinbarungAuswaehlenAA(
                                          SeminarisFrame parent);
   AuswaehlenListeAA getSVAuswaehlenAA( SeminarisFrame parent);
   AuswaehlenListeAA getSVAuswaehlenAA(
                                    SeminarisFrame parent, String policy);
   ILeitungsauftragAuswaehlenAA getLeitungsauftragAuswaehlenAA(
                                    SeminarisFrame parent);
   AuswaehlenListeAA getSeminartypAuswaehlenAA( SeminarisFrame parent);
}
```

Hier zeigen sich leider erste Probleme: Entgegen den internen Designvorgaben der Gruppe wurde die Klasse `DozentenvereinbarungAuswaehlenAA` nicht von der im Grobentwurf hierfür festgelegten Templateklasse `AuswaehlenListeAA` abgeleitet. Diese zusätzliche Indirektion ist aber Voraussetzung dafür, die aufrufende Klasse absolut frei von der Abhängigkeit zu konkreten Auswahlklasse zu bekommen. Gelingt dies nicht, so verbleibt zumindest noch eine Typabhängigkeit bei der aufrufenden Klasse:

```
DozentenvereinbarungAuswaehlenAA dvAuswaehlenAA =
                              Factory.getDozentenvereinbarungAuswaehlenAA();
```

Das nachträgliche Ableiten würde die zusätzliche Implementierung der abstrakten Methoden von `AuswaehlenListeAA` bedeuten, die derzeit noch nicht vorhanden sind. Einfacher ist es ein Interface `IDozentenvereinbarung` einzuführen, in dem die von den aufrufenden Instanzen benötigten Methoden definiert werden.

Die Klassen `PersonAuswaehlenAA` und `LeitungsauftragAuswaehlenAA` leiten sich zwar von der abstrakte Klasse `AuswaehlenListeAA` ab, bieten jedoch den aufrufenden Klassen noch weitere Methoden an, die nicht in der abstrakten Klasse definiert sind. Um auch hier die angesprochenen Typabhängigkeiten zu vermeiden, müssen auch für diese Klassen Interfaces geschaffen werden.

Die Klasse `SVAuswaehlenAA` bietet noch eine zweite Methode zur Erzeugung einer Auswahlinstanz an. Diese Methode erhält im Originalcode als zweiten Parameter eine *Policy*-Klasse übergeben, bei der es sich um eine von sechs möglichen inneren Klassen der Klasse `SVAuswaehlenAA` handelt. Um diese Abhängigkeit auf möglichst einfache Weise zu beseitigen, wird der Parametertyp in `String` umgewandelt und die Zuweisung der aktuellen *Policy* erfolgt über Fallunterscheidung in der Erzeugermethode.

Die Implementierung der `AAKlassenFabrik` erfolgt trivialerweise durch Erzeugung und Rückgabe der Auswahlklassen. Das Erzeugen der Fabrik erfolgt beim Start der Anwendung in der Hauptschnittstellenklasse `SeminarisH`. Im ersten Ansatz der Refaktorisierung erfolgt die Verwaltung der Fabrik in einer neu geschaffenen Systemklasse `SeminarisSystem`:

```
public class SeminarisSystem {
    private static IAAKlassenFabrik aaKlassenFabrik = null;

    public static IAAKlassenFabrik getAAKlassenFabrik(){
        if( aaKlassenFabrik == null)
            throw new IllegalStateException(
                                    "AAKlassenFabrik ist nicht gesetzt.");
        return aaKlassenFabrik;
    }

    public static void setAAKlassenFabrik(
                                IAAKlassenFabrik aktuelleAAKlassenFabrik){
        aaKlassenFabrik = aktuelleAAKlassenFabrik;
    }
}
```

Dieser Ansatz der zentralen Verwaltung erweist sich im Nachfolgenden als ungünstig. Eine Erläuterung bringt der nachfolgende Abschnitt.

Ein grundsätzliches Problem an der vorgestellten Lösung ist ferner, dass das Erzeugen von Auswahlinstanzen nach wie vor über den direkten Zugriff auf die Konstruktoren und sonstige Erzeugemethoden der Auswahlklassen erfolgen kann. Zur zwingenden Verwendung der Fabrik für die Erzeugung der Instanzen müsste die Paketstruktur dahingehend geändert werden, dass alle Auswahlklassen exklusiv mit der Fabrik im selben Verzeichnis liegen. Dann könnte für alle Konstruktoren der Auswahlklassen paketlokaler Zugriff gesetzt werden und die Erzeugung einer Auswahlklasse auf direktem Wege von außerhalb des Paketes wäre unterbunden.

11.7.3 Zwischenergebnisse

Ein Blick auf die Projektmetriken vermittelt den Eindruck, dass die Refaktorisierung absolut kontraproduktiv war:

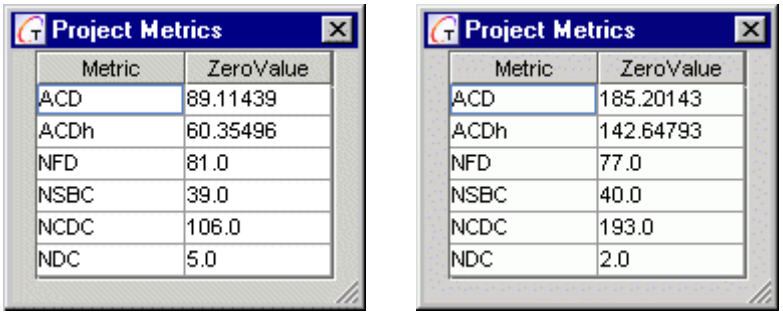


Abbildung 124: Vergleich Projektmetriken vor und nach Einführung einer Schnittstellen-Fabrik

Die durchschnittliche Abhängigkeit einer Komponente ist um 109 Prozent angestiegen, beschränkt auf fest verdrahtete Abhängigkeiten sogar um 138 Prozent. Fünf kleinere Abhängigkeitszyklen sind zu zwei größeren Zyklen verschmolzen, die Anzahl der darin befindlichen Komponenten ist um 82 Prozent gestiegen.

Was sind die Gründe für diese unerwartete Veränderungen der Metriken? Zwei Aspekte sind für den Anstieg bedeutsam. Der erste wird augenfällig, wenn wir einen Ausschnitt aus dem Klassendiagramm betrachten, der die Verwendung der neu eingeführten Fabrik veranschaulicht:

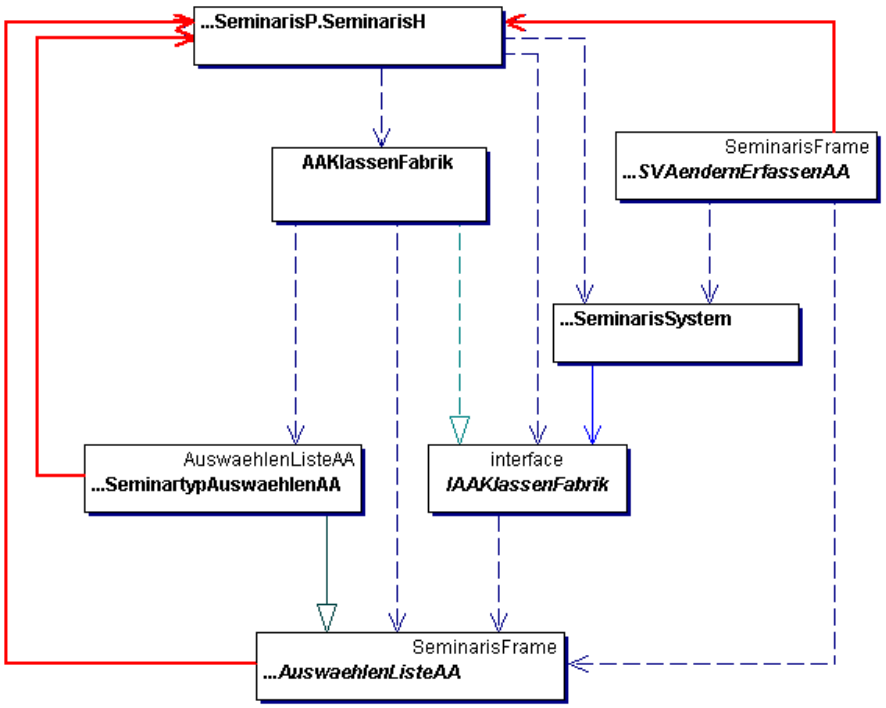


Abbildung 125: Verwendung der Schnittstellen-Fabrik in SVAendernErfassenK zur Erzeugung der Auswahlklasse SeminartypAuswaehlenAA

Sämtliche Schnittstellenklassen greifen auf das statische Attribut `MainWindow` der Hauptschnittstellenklasse `SeminarisH` zu, um in diesem Hauptfenster (eine Instanz der Klasse `SachbearbeiterA`) Hinweise und Fehlermeldungen anzuzeigen. Dadurch wird die komplette Schnittstellenschicht zyklisch verbunden.

Diese zyklische Verbindung war vor der Refaktorisierung deshalb nicht erkennbar, weil durch die Reflection-Mechanismen in der Klasse `SachbearbeiterA` zur Erzeugung der Auswahlschnittstellen der Zyklus durchbrochen war. Die Erzeugung einer Instanz der `AAKlassenFabrik` in der Hauptschnittstellenklasse schließt nun den Zyklus durch die dadurch entstehende Abhängigkeit von `SeminarisH` zu den Auswahlschnittstellen.

Dies wird durch die ermittelten Abhängigkeitsmetriken und insbesondere durch die Komponentenmetriken deutlich:

Dependent class	Dependee class	dep_comp	rACD	rACDh	rNFD	rNSBC	rNCDC	rNDC	sub_hw	d_type
Dozentenvereinbarung	DVKostenBerechnenK	1	45,3	48,4	5,2	13	35,8	-50	11	create and static access
SeminarisH	AbstractAAKlassenFabrik	1	36,8	61,9	7,8	10	36,3	-50	33	create and static access
AbstractAAKlassenFabrik	AAKlassenFabrik	1	36,4	61,9	6,5	10	35,8	-50	100	create dependency
AAKlassenFabrik	BelegungAuswaehlenAA	1	10,5	12	6,5	2,5	11,9	0	100	create and static access
SeminarisK	SeminarisDatenbank	1	5,45	14,8	1,3	15	2,07	-50	20	static access
AAKlassenFabrik	DozentenvereinbarungAuswaehlenAA	1	5,4	5,39	-22	-23	0,52	-50	100	create and static access
AAKlassenFabrik	SeminartypAuswaehlenAA	1	4,6	5,55	-21	-20	5,18	0	100	create and static access
AAKlassenFabrik	LeitungsauftragAuswaehlenAA	1	4,6	6,05	-21	-20	5,18	0	100	create and static access
DruckDokumentDK	SeminarisK	1	2,41	3,53	1,3	2,5	1,04	0	100	static access
BelegungAuswaehlenAA	BelegungStornierenAA	1	2,31	2,53	1,3	0	2,59	0	33	create and static access
BelegungAuswaehlenAA	TeilnehmerAnmeldenAA	1	2,31	2,53	1,3	2,5	2,59	0	33	create and static access
TeilnehmerAnmeldenAA	TeilnehmerAnmeldenK	1	1,85	2,03	1,3	2,5	2,07	0	25	create and static access
BelegungStornierenAA	BelegungStornierenK	1	1,85	2,03	1,3	0	2,07	0	50	create and static access
SVAuswaehlenAA	FSVErfassenAA	1	1,85	2,02	-7,8	0	2,07	0	33	create and static access
BelegungAuswaehlenAA	BelegungAendernAA	1	1,85	2,02	1,3	2,5	2,07	0	50	create and static access

Abbildung 126: Abhängigkeitsmetriken nach Einführung der Schnittstellenfabrik

Während in den Abhängigkeitsmetriken noch kein Hinweis auf die aus der Schnittstellenschicht zu `SeminarisH` führenden Abhängigkeiten zu finden ist, liefern die Komponentenmetriken einen eindeutigen Anhaltspunkt in Gestalt der Metrik rACD_in:

Component	cd	cd_h	cd_tr	cd_h_tr	rACD_in	rACD_out
SeminarisH	5	4	213	194	53,76	53,71
Dozentenvereinbarung	14	7	213	194	45,52	45,46
DVKostenBerechnenK	6	5	213	194	45,32	45,07
AbstractAAKlassenFabrik	2	1	213	194	36,75	36,7
AAKlassenFabrik	14	8	213	194	36,44	36,39
SeminarisDatenbank	12	7	213	194	16,95	16,6
SVAuswaehlenAA	13	12	213	194	11,88	11,83
BelegungAuswaehlenAA	13	11	213	194	10,54	10,49
SeminarisK	6	6	213	194	6,555	6,505
DozentenvereinbarungAuswaehlenAA	7	2	213	194	5,4	5,349
SeminartypAuswaehlenAA	11	10	213	194	4,603	4,553
LeitungsauftragAuswaehlenAA	13	10	213	194	4,603	4,553
DVAktionenAA	16	9	213	194	3,224	3,174
TeilnehmerAnmeldenAA	9	7	213	194	2,307	2,257
BelegungStornierenAA	8	3	213	194	2,307	2,257

Abbildung 127: Komponentenmetriken nach Einführung der Schnittstellenfabrik

11.7.4 Nachbesserung

Letztlich resultiert die Lösung des Problems wieder im Schaffen eines globalen abhängigkeitsreduzierten Zugriffs auf das Hauptfenster der Anwendung. Wir orientieren uns zunächst am Vorgehen aus Abschnitt 11.2 und verwalten das Hauptfenster in der im ersten Durchgang eingeführten zentralen Klasse `SeminarisSystem`:

```
public class SeminarisSystem {
    private static IAAKlassenFabrik aaKlassenFabrik = null;
    private static JFrame hauptFenster = null;

    public static IAAKlassenFabrik getAAKlassenFabrik(){
        ...
    }

    public static void setAAKlassenFabrik(
                                IAAKlassenFabrik aktuelleAAKlassenFabrik){
        ...
    }

    public static JFrame getHauptFenster() {
        if( hauptFenster == null)
            throw new IllegalStateException(
                                        "Hauptfenster ist nicht gesetzt.");
        return hauptFenster;
    }

    public static void setHauptFenster( JFrame aktuellesHauptFenster) {
        hauptFenster = aktuellesHauptFenster;
    }}
```

Dieser Entwurf führt jedoch dazu, dass jede Klasse, die sich zur Anforderung der Instanz des Hauptfensters der Klasse `SeminarisSystem` bedienen muss, vom Interface `IAAKlassenFabrik` und transitiv von den darin definierten Typen abhängig wird. Damit wird insbesondere die durch den Factory-Mechanismus gewonnene Unabhängigkeit in der Testphase wieder deutlich reduziert.

Es sollte daher auf jeden Fall von einer zentralen Instanz zur Verwaltung von Ressourcen (wie z.B. Fabriken, Datenbanken, Anwendungsfenster) abgesehen werden. Stattdessen erhält jede dieser Ressourcen eine eigene „Provider“-Klasse zur Seite gestellt (siehe auch Abschnitt 11.5.2.5). Dies führt zur Einführung von zwei Klassen `AAKlassenFabrikProvider` und `MainWindowProvider`, die beide den ihnen zugehörigen Part aus der oben beschriebenen Klasse `SeminarisSystem` übernehmen.

Der gesamte Zeitaufwand für die Refaktorisierung beträgt 4 Stunden.

11.7.5 Ergebnisse

Man erhält folgende Veränderung der Projektmetriken:

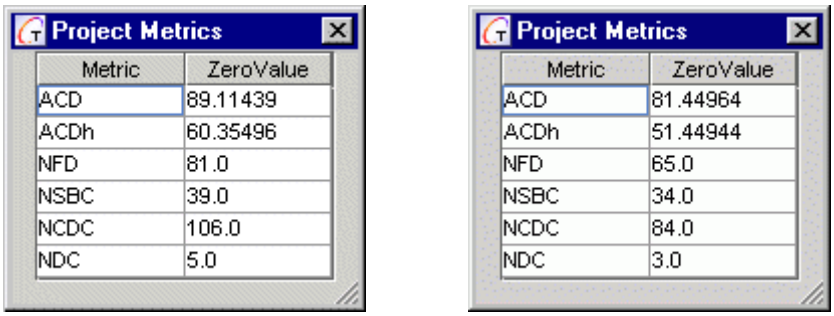


Abbildung 128: Projektmetriken vor und nach abschließender Realisierung der AAKlassenFabrik

Die Anzahl der Klassen, von denen eine Klasse im Durchschnitt direkt oder indirekt abhängig ist, ist um 8,2 Prozent gesunken, beschränkt auf fest verdrahtete Abhängigkeiten um immerhin 14,2 Prozent.

Die Zahl der Abhängigkeitszyklen hat sich echt von fünf auf drei reduziert (keine Verschmelzung), die Anzahl der zyklischen abhängigen Komponenten hat sich dadurch um 20,8 Prozent verringert.

Ein Blick auf die Abhängigkeits- und Komponentenmetriken lässt erkennen, dass insbesondere an den ersten 50 Abhängigkeiten mit dem höchsten rACD-Wert (>0,48 Prozent) keine Auswahlklassen mehr beteiligt sind.

Teil IV - Resümee

Der letzten Teil dieser Arbeit umfasst eine Zusammenfassung und Bewertung der wichtigsten Ergebnisse der Arbeit. Daran schließt sich ein Ausblick auf notwendige und wünschenswerte Arbeiten im Rahmen der Weiterführung des Themas an.

12 Zusammenfassung

Mit *Design2Test* steht, als Ergebnis des ersten Teils dieser Arbeit, Softwareentwicklern künftig ein Werkzeug zur Testbarkeitsanalyse als integrierter Bestandteil einer Entwicklungsumgebung zur Verfügung. Die Testbarkeitsanalyse umfasst Aussagen über aus Sicht der Testbarkeit interessierende Eigenschaften eines Softwaresystems und liefert insbesondere Hinweise auf mögliche testkritische Abhängigkeiten zwischen einzelnen Komponenten. Testkritische Abhängigkeiten sind Abhängigkeiten, die einen potentiell hohen Einfluss auf die Testbarkeit besitzen. Ihre Untersuchung erscheint zur Verbesserung der Testbarkeit besonders lohnenswert.

Mit *Design2Test* erhält der Entwickler eine vollständige Übersicht der zwischen den Klassen eines Java-Programms vorhandenen Abhängigkeiten einschließlich der Anzeige des zur Abhängigkeit führenden Programmcodes. Für jede Klasse erfolgt eine Aufschlüsselung nach ein- und ausgehenden Abhängigkeiten. Zur Messung der Unabhängigkeit der Systemkomponenten stehen Metriken zur Verfügung, die beispielsweise die durchschnittliche Abhängigkeit der Komponenten untereinander oder die Anzahl und Größe von Abhängigkeitszyklen messen. Zur Identifizierung testkritischer Abhängigkeiten werden für jede existierende Abhängigkeit Reduktionsmetriken berechnet, deren Wert anzeigt, wie sich die Entfernung der Abhängigkeit auf die Metrikwerte des Gesamtsystems auswirken würde.

Unter Verwendung von *Design2Test* erfolgte im zweiten Teil dieser Arbeit die Testbarkeitsanalyse eines Softwaresystems (*SeminarIS*). Auf Grundlage der im Rahmen der Analyse errechneten Metriken wurden aus *SeminarIS* jene 30 testkritischen Abhängigkeiten für eine Untersuchung und Bewertung ausgewählt, deren Entfernung den größten Beitrag zur Verbesserung der Unabhängigkeit der Komponenten leistet. Zur Bestimmung des Grads der Unabhängigkeit der Komponenten wurde die Metrik ACD verwendet, die die Anzahl der Klassen angibt, mit denen jede Klasse des Systems im Durchschnitt direkt oder indirekt über Abhängigkeiten verbunden ist. Mit Hilfe der Reduktionsmetrik rACD konnte dann die Identifizierung der 30 zu untersuchenden testkritischen Abhängigkeiten erfolgen.

Die Analyse dieser testkritischen Abhängigkeiten zeigte, dass *SeminarIS* unter einigen massiven Testproblemen litt. Beinahe alle untersuchten Abhängigkeiten wiesen auf eines oder auch mehrere dieser Probleme hin. Die aufgedeckten Testprobleme waren bedingt durch ungünstige Entwurfsentscheidungen oder Implementierungsfehler bei der Umsetzung des Entwurfs. Dies wurde zum Anlass genommen, eine nähere Untersuchung der Entwurfsprobleme vorzunehmen. Daraus resultierte eine eingehende Beschreibung der Probleme und ihrer konkreten Auswirkungen auf die Testbarkeit von *SeminarIS*:

- Die Verletzung von Hierarchien in Schichtenstrukturen führt über das Entstehen von Abhängigkeitszyklen zu einem immensen Anstieg der Abhängigkeiten. *SeminarIS* ist hiervon besonders stark und in verschiedensten Varianten betroffen. Lässt man einen ohne Not an einer Stelle eingefügten *Reflection*-Mechanismus außer Betracht, der

zwar die syntaktische aber nicht die semantische Abhängigkeit verhindert, so sind 225 der insgesamt 278 *SeminarIS*-Klassen in einem einzigen Zyklus voneinander abhängig. Alle Verletzungen der Schichtenhierarchien konnten als unnötig und weitestgehend durch Implementierungsfehler bedingt nachgewiesen werden.

- Die Realisierung des Zugriffs auf globale Basiskomponenten, insbesondere der Zugriff auf die Datenbank, führte in *SeminarIS* zu massiven Testproblemen. So kann bei der vorhandenen Implementierung beispielsweise nicht verhindert werden, dass auch beim Test der meisten Benutzerschnittstellenklassen stets die Datenbank gestartet, bereits abgefragt und unter Umständen durch das verwendete Persistenz-Rahmenwerk zusätzlich benötigter Code (Proxyklassen) generiert und kompiliert wird.

- Die meisten der als testkritisch analysierten Abhängigkeiten wiesen auf ein Entwurfsproblem in der Datenhaltungsschicht von SeminarIS hin. Dort wurden die Domänenklassen als Fassadenklassen für den Zugriff auf die komplette Datenhaltungsschicht realisiert. Dies führte bereits bei der Implementierung des Systems zu Problemen und zum erforderlichen Einsatz eines *workarounds*, also zusätzlichem Code zur Umgehung eines sonst nicht vermeidbaren Fehlers. Unter dem Gesichtspunkt der Testbarkeit konnte dieser Entwurfsfehler als besonders problematisch dargestellt werden. Der ohnehin schon zusätzliche Implementierungsaufwand aufgrund des gewählten Entwurfsmusters führt zu einer Verdopplung des Testaufwands durch die notwendige Implementierung redundanter Testfälle für die überwiegende Zahl der Domänenklassenmethoden.

- Ein weiteres Testproblem resultierte in der Schnittstellenschicht aus der Realisierung der Beziehungen zwischen Anwendungsfällen (*include* und *extend*). Der Aufruf eines anderen Anwendungsfalles wurde über die Erzeugung einer Instanz der benötigten Schnittstellenklasse verwirklicht. Dadurch entstanden fest verdrahtete Abhängigkeiten zwischen den Schnittstellenklassen der beteiligten Anwendungsfälle. War in der Anforderungsspezifikation ein wechselseitiger Aufruf vorgesehen, resultierte daraus die zyklische Abhängigkeit der Anwendungsfallklassen. Dies führte dazu, dass der unabhängige Test der Anwendungsfälle nicht mehr durchgeführt werden konnte.

Die Hauptursache für das Entstehen der angehäuften Testprobleme lag in der Vorgehensweise bei der Entwicklung von *SeminarIS*. Ein isolierter Test von kleineren Teileinheiten des Systems war im Entwicklungsprozess nicht vorgesehen. Dies führte in der Konsequenz dazu, dass weder bei Architekturentscheidungen in der Entwurfsphase noch während der Implementierungsphase Testbarkeitsaspekte Berücksichtigung fanden

Einige der erkannten Entwurfsprobleme konnten mit minimalem Aufwand beseitigt werden. Selbst bei den Problemen, die eine umfassende Veränderung in der Implementierung erforderlich machten, war der Aufwand im Verhältnis zur Gesamtentwicklungszeit gering. Neben den nachfolgend noch beschriebenen konkreten Verbesserungen, die durch die Refaktorisierung im Hinblick auf die Testbarkeit von *SeminarIS* erreicht werden konnten, lassen auch die im Anschluss gemessenen Metrikwerte (mit einer Ausnahme) auf eine Verbesserung der Testbarkeit schließen.

Nachfolgend eine Auflistung der wichtigsten Refaktorisierungsschritte:

- Im Rahmen der Realisierung eines veränderten Zugriffs auf globale Basiskomponenten wurden verschiedene, in der Literatur zu findende Möglichkeiten zur Lösung dieses Problems diskutiert. Keine der Lösungen konnte letztlich aus Sicht der Testbarkeit vollständig überzeugen, so dass ein eigenständiges Konzept für den Umgang mit Basiskomponenten entwickelt und vorgeschlagen wurde. Auffällig an

dieser Lösung war, dass sie zwar zu keiner Verbesserung der Metrikwerte führte, die erreichte Konfigurierbarkeit der Basiskomponenten aber dennoch einen deutlichen Vorteil bei der Testdurchführung darstellt, da damit im Rahmen der Initialisierung von Testfällen der Austausch der Komponenten durch Teststellvertreter ermöglicht wird.

- Die Refaktorisierung der Datenhaltungsschicht war zwar mit dem meisten Aufwand verbunden, führte aber dafür zu einer Verringerung des Wertes der ACD-Metrik um über 40 Prozent. War vor der Refaktorisierung eine Klasse im Durchschnitt von ca. 89 anderen Klassen direkt oder indirekt abhängig, so sank diese Zahl nach der Refaktorisierung auf ca. 51 Klassen. Die durchschnittliche Zyklengröße sank durch die Umgestaltung der Datenhaltung von ca. 21 Klassen auf ca. 8 Klassen. Im Hinblick auf den zu leistenden Testaufwand war insbesondere die Tatsache von Bedeutung, dass durch die Refaktorisierung die Anzahl der Methoden in der Datenhaltung um über ein Drittel von 964 auf 634 Methoden gesenkt werden konnte. Damit wird der an dieser Stelle zu leistende Testaufwand deutlich reduziert.

- Zur Umgestaltung der Schnittstellenschicht wurde ein *Factory*-Mechanismus eingeführt, der eine Trennung von Objekterzeugung und Verwendung der Instanzen ermöglichte. Damit konnte die Unabhängigkeit der Anwendungsfälle erreicht und alle zyklischen Abhängigkeiten aus der Schnittstellenschicht beseitigt werden, was zu einer Reduzierung der insgesamt an Zyklen beteiligten Klassen um etwa 20 Prozent führte.

Insgesamt zeigte sich, dass die mit Hilfe von *Design2Test* ermittelten Metriken ein hervorragendes Instrument zur Anzeige von massiven Testproblemen waren. Wie gezeigt werden konnte, resultierten diese Schwierigkeiten zur Testfallerstellung und Testdurchführung aus teilweise schwerwiegenden Entwurfsproblemen und Implementierungsfehlern. Über die Beseitigung der von *Design2Test* identifizierten testkritischen Abhängigkeiten konnte in der Folge eine deutliche Verbesserung der Testbarkeit des analysierten Systems erreicht werden. Zudem konnte gezeigt werden, dass die Entfernung dieser Probleme mit vergleichsweise geringem Aufwand erfolgen kann.

Abweichend von der Aufgabenstellung konnte die vorgesehene Testfallerstellung vor und nach der Refaktorisierung des Systems und die Bestimmung der jeweiligen Testaufwände aus Zeitgründen nicht mehr in sinnvoller Weise durchgeführt werden. Dies lag zum einen daran, dass die Erstellung von *Design2Test* deutlich mehr Zeit erforderte als geplant. Zum anderem waren durch die vorhandenen Testprobleme die Voraussetzungen für die Erstellung und Durchführung von möglichst unabhängigen *Unit*-Tests denkbar schlecht.

13 Ausblick

Der empirische Nachweis, dass mit den durchgeführten Refaktorisierungsschritten nicht nur eine qualitative Verbesserung der Testbarkeit, sondern auch eine tatsächliche Verringerung der Testaufwände verbunden ist, konnte wie beschrieben aus Zeitgründen nicht mehr geführt werden. Dies bleibt weiteren Untersuchungen vorbehalten.

Ferner wurde im Rahmen dieser Arbeit für die Analyse von testkritischen Abhängigkeiten nur ein kleiner Teil der in *Design2Test* verfügbaren Metriken und Informationen ausgewertet. Dies führte in erster Linie zur Aufdeckung von Testproblemen, die auf allgemeinen Entwurfsfehlern basierten. Hier ist die Frage interessant, welche zusätzlichen Testprobleme mit der noch nicht ausgenutzten Funktionalität von *Design2Test* entdeckt werden können. Möglicherweise wäre für die Untersuchung dieser Frage auch ein Softwaresystem besser geeignet, das unter weniger gravierenden Entwurfsmängeln leidet, um in diesem Zusammenhang den Einfluss von allgemeinen Entwurfsproblemen auf die Testbarkeit gering zu halten.

In der aktuell verfügbaren Version wurde *ImproveT* um die Fähigkeit erweitert, aus den Daten des Abhängigkeitsgraphen Informationen über polymorphe Abhängigkeiten zu ermitteln. Diese Information wurde bei der Erstellung der vorliegenden Arbeit noch nicht berücksichtigt. Polymorphie und dynamische Bindung erschweren den Test objektorientierter Programme, da sie zu einer Vermehrung der potenziellen Ablaufpfade führen und damit zu einer Vermehrung der potenziellen Fehler **[**SnWi01**]**. Die Anzeige der polymorphen Abhängigkeiten kann eine Einschätzung dieses Problems für ein konkretes System ermöglichen.

Im Rahmen dieser Arbeit war ursprünglich geplant, über *Design2Test* die Metrikberechnung auch für Abhängigkeiten auf Paketebene anbieten zu können. *ImproveT* stellt hierfür bereits grundsätzliche Funktionalität zur Verfügung. Allerdings ist hier noch die Frage der genauen Zuordnung von Klassen zu Paketstrukturen zu klären. Dies war auch der Grund dafür, dass die Umsetzung in *Design2Test* nach einer bereits vorhandenen ersten prototypischen Realisierung wieder zurückgestellt wurde.

Literaturverzeichnis

[Beck00**]** BECK, Kent: Extreme Programming Explained: Embrace Changed. Addison-Wesley, 2000.

[Beiz94**]** BEIZER, Boris: *Testing Technology – The Growing Gap*. American Programmer, Jahrgang 7, Nr.4, 1994.

[Bind94**]** BINDER R. V.: *Design for Testability in Object-Oriented Systems*. Communications of the ACM, vol. 37, pp.87-101, Sept. 1994.

[Bind99**]** BINDER, Robert: *Testing Object-Oriented Systems*. Addison-Wesley, 1999.

[Coop98**]** COOPER, James W.: *The DesignPatterns Java Companion.* Addison-Wesley, Design Pattern Series, 1998.

[DoFr94**]** DOONG, R.-K; FRANKL, P. G.: *The ASTOOT Approach to Testing Object-Oriented Programs*. ACM Transactions on Software Engineering and Methodologie, pp. 101-130, 1994.

[Gamm94**]** GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994

[GrSy01**]** MCGREGOR, John D.; SYKES, David A.: *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley, 2001.

[IEEE90**]** IEEE STD 610.12-1990: *IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[ISO9126**]** ISO/IEC 9126: *Software Engineering - Product quality*.

[Jung02**]** JUNGMAYR, Stefan: *Testability measurement and software dependencies*. In Proceedings of the 12th International Workshop on Software Measurement, Magdeburg, Germany, pp. 179-202, October 7-9, 2002.

[Lako96**]** LAKOS, J.: *Large-scale C++ software design.* Addison-Wesley, 1996.

[Link01**]** LINK , Johannes: *Unit Tests mit Java: Der Test-first-Ansatz*. dpunkt-Verlag, Heidelberg, 2001.

[LoSh98**]** LO, B.W.N.; SHI, H.: *A preliminary testability model for object-oriented software*. Proceedings of the 1998 International Conference Software Engineering: Education and Practice, Dunedin, New Zealand (M. Purvis, ed.), pp. 330-337, 1998.

[OUML00**]** OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language Specification Version 1.3*. Object Management Group Inc., 2000.

[SnWi01**]** SNEED, Harry M.; WINTER, Mario: *Testen objektorientierter Software – Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme.* Carl Hanser Verlag, München, 2001.

[SWEII99**]** SIX, Hans-Werner; WINTER, Mario, et al.: *Software Engineering II – Objektorientierte Softwareentwicklung*. Kurs 1794, FernUniversität Hagen – Gesamthochschule in Hagen, Fachbereich Informatik, 1999.

[ToSZ02**]** TOGETHERSOFT GMBH: *Softwareentwicklung in einer neuen Zeit – Unternehmensporträt TogetherSoft GmbH* – URL: http://www.Togethersoft.de/html/press/Corporate_Backgrounder_Final.doc (2002-07-05).

[ToUG02**]** *Together*SOFT CORPORATION: *UserGuide Together ControlCenter Version 6.0*. *Together*Soft, 2001. – Anwenderhandbuch zum *Together*Soft *ControlCenter* Version 6.0 in der Fassung vom 11. April 2002.

[Wint00**]** WINTER, M.: *Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und regressionstest.* Informatik Forschung und Entwicklung, Vol. 15, Nr. 3, pp. 121-132, 2000.

[Wulff02**]** WULFF, Nikolaus: *Wer misst, misst Mist - Metriken zur Steigerung der Softwarequalität*. JavaMagazin, Ausgabe 04.2002, pp. 17-23, April 2002.

Anhang A Anwendungsfälle Testbarkeitsanalyse

Inhalt:

A.1 ANWENDUNGSFALLDIAGRAMM 133

A.2 TEXTUELLE ANWENDUNGSFALLSPEZIFIKATIONEN 134

A.2.1 ANWENDUNGSFALL „TESTBARKEITSMETRIKEN BERECHNEN“ 134
A.2.2 ANWENDUNGSFALL „TESTBARKEITSMETRIKEN ERSTMALIG BERECHNEN“ 135
A.2.3 ANWENDUNGSFALL „METRIKEN AUSWÄHLEN“ 136
A.2.4 ANWENDUNGSFALL „SUCHOPTIONEN EINSTELLEN“ 136
A.2.5 ANWENDUNGSFALL „NAVIGATION ZUM SOURCECODE“ 136
A.2.6 ANWENDUNGSFALL „NAVIGATION ZU KLASSENDIAGRAMM“ 137
A.2.7 ANWENDUNGSFALL „METRIKEN SPEICHERN“ 137
A.2.8 ANWENDUNGSFALL „METRIKEN DRUCKEN“ 137
A.2.9 ANWENDUNGSFALL „ABHÄNGIGKEITSGRAPH ANZEIGEN“ 138
A.2.10 ANWENDUNGSFALL „GRAPH SPEICHERN“ 138
A.2.11 ANWENDUNGSFALL „GRAPH DRUCKEN“ 139

A.1 Anwendungsfalldiagramm

Die nachstehende Abbildung zeigt das Anwendungsfalldiagramm für die Together-Erweiterung zur Testbarkeitsanalyse:

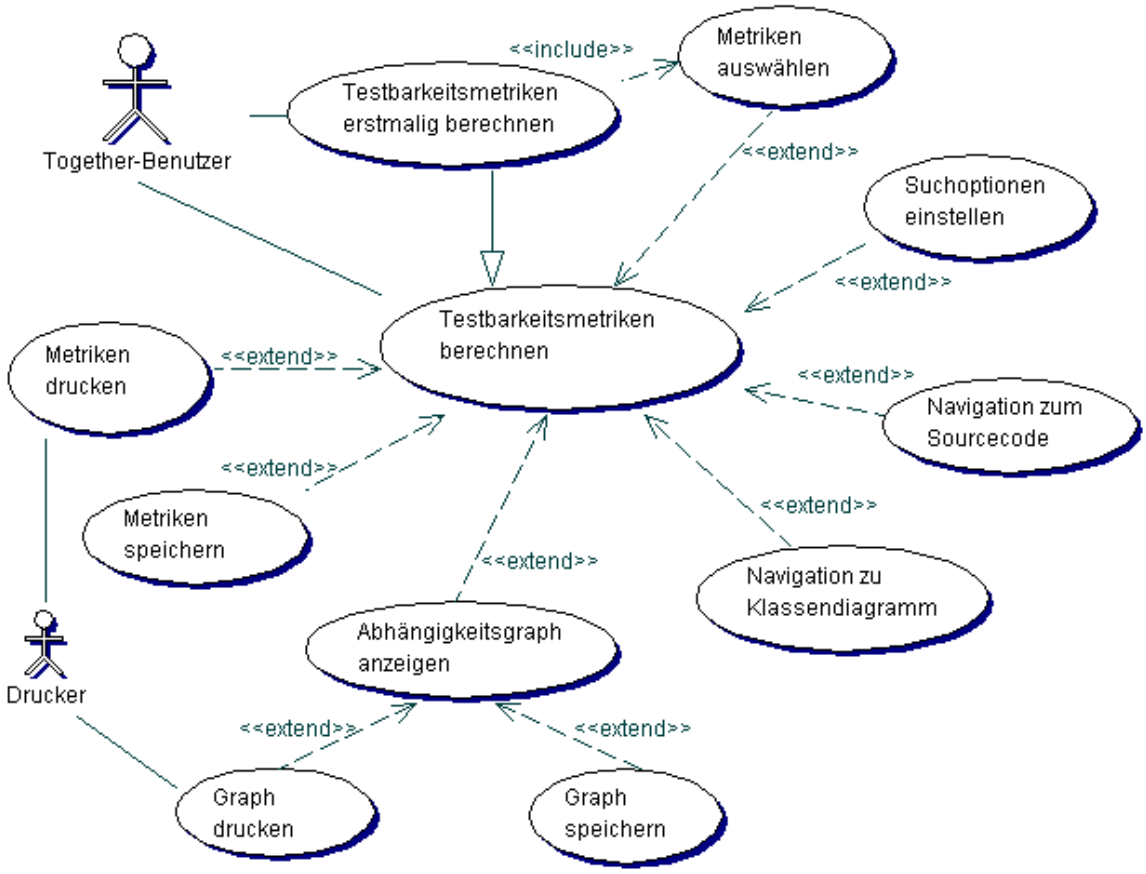


Abbildung A1: Anwendungsfalldiagramm Testbarkeitsanalyse

Die textuelle Beschreibung der Anwendungsfälle erfolgt in den folgenden Abschnitten.

A.2 Textuelle Anwendungsfallspezifikationen

A.2.1 Anwendungsfall „Testbarkeitsmetriken berechnen“

use case Testbarkeitsmetriken berechnen
actors Together-Benutzer
precondition

Das Together ControlCenter ist gestartet. Die Together-Erweiterung für die Testbarkeitsanalyse von Abhängigkeiten wurde beim Start geladen. Zum Starten des Moduls wurde ein Menüpunkt in das Hauptmenü integriert. Ein Together-Projekt ist geöffnet.

main flow

Der Benutzer startet die Testbarkeitsanalyse über das Hauptmenü der Together-Anwendung oder über das Kontextmenü einer bereits vorhandenen Metrikanzeige. Die Analyse erfolgt auf Basis der zuletzt ausgewählten Metriken und der aktuell eingestellten Suchoptionen. Die ermittelten Abhängigkeiten und alle Komponenten des Projekts werden gemeinsam mit den berechneten Testbarkeitsmetriken in Tabellenform angezeigt. In der Anzeige der Abhängigkeitsmetriken werden zu jeder auf Klassenebene existierenden Abhängigkeit die berechneten Abhängigkeitsmetriken angezeigt. Selektiert der Benutzer eine der Abhängigkeiten, so werden die Stellen des Sourcecodes angezeigt, die Auslöser der Abhängigkeit sind. In der Anzeige der Komponentenmetriken werden zu jeder im Projekt existierenden Komponente (Klasse) die berechneten Komponentenmetriken angezeigt. Selektiert der Benutzer eine der Komponenten, so werden die Komponenten angezeigt, mit denen die Komponente in Abhängigkeit steht. Dabei wird unterschieden nach Klassen, die von der ausgewählten Komponente abhängig sind und Klassen, von denen die Komponente selbst wiederum abhängig ist.

alternate flow Metriken auswählen

Der Benutzer möchte die zu berechnenden Testbarkeitsmetriken verändern. Er erhält die für die Analyse verfügbaren Testbarkeitsmetriken erneut zur Auswahl angeboten und kann eine neue Metrikmenge zusammenstellen (extension point „Metriken auswählen“).

alternate flow Suchoptionen einstellen

Der Benutzer verändert die Einstellung der Programmkomponenten (Pakete, Klassen), die bei der Suche nach Abhängigkeiten durchlaufen und als Ziel von Abhängigkeiten berücksichtigt werden sollen (extension point „Suchoptionen einstellen“).

exceptional flow Abbruch der Metrikberechnung

Der Benutzer kann die Testbakeitsanalyse während der Berechnung der Testbarkeitsmetriken jederzeit abbrechen.

alternate flow Anzeige der Komponentenmetriken

Der Benutzer wechselt ausgehend von einer an einer Abhängigkeit beteiligten Klasse aus der Anzeige der Abhängigkeitsmetriken zur Anzeige der Komponentenmetriken dieser Klasse.

alternate flow Anzeige der Abhängigkeitsmetriken

Ausgehend von einer über eine Abhängigkeit verbundene Komponente kann der Benutzer aus der Anzeige der Komponentenmetriken zur Anzeige der zugrunde liegenden Abhängigkeitsmetriken wechseln.

alternate flow Navigation zum Source-Code

Der Benutzer kann direkt an die Stellen im Programmtext springen, die Auslöser einer Abhängigkeit sind. Ferner können alle in den Metriktabellen angezeigten Klassen im Texteditor von Together geöffnet werden (extension point „Navigation zum Sourcecode“).

alternate flow Navigation zu Klassendiagramm

Der Benutzer lässt sich zu einer beliebigen Klasse das zugehörige Klassendiagramm anzeigen (extension point „Navigation zu Klassendiagramm“).

alternate flow Testbarkeitsmetriken erneut berechnen

Der Benutzer startet eine erneute Berechnung der Metriken..

alternate flow Anzeige von Hilfetexten zu Metriken

Der Benutzer ruft zu den angezeigten Metriken in einem Hilfefenster nähere Erläuterungen ab.

alternate flow Speichern der Metriken

Die errechneten Metriken können im Textformat zur weiteren Verarbeitung (z.B. mit einem Tabellenkalkulations- oder Statistikprogramm) gespeichert werden (extension point „Metriken speichern“).

alternate flow Ausdruck der Metriken

Der Benutzer lässt sich die tabellarische Anzeige der Metriken ausdrucken (extension point „Metriken drucken“).

alternate flow Graphische Darstellung der Abhängigkeiten

Der Benutzer ruft eine Darstellung der Komponenten des Projektes und der zwischen ihnen ermittelten Abhängigkeiten als geometrischer Graph auf (extension point „Abhängigkeitsgraph anzeigen“).

alternate flow Anzeige der Projektmetriken

In der Anzeige der Abhängigkeitsmetriken ruft der Benutzer die für das Gesamtprojekt errechneten Reduktionsmetriken zur Anzeige auf.

alternate flow Anzeige der Projektmetriken nach Ausschluss

Der Benutzer wählt in der Anzeige der Abhängigkeitsmetriken eine oder mehrere Abhängigkeiten aus und erhält die Projektmetriken angezeigt, die sich nach Entfernen der ausgewählten Abhängigkeiten ergeben.

postcondition

Die vom Benutzer ausgewählten Testbarkeitsmetriken wurden berechnet und werden angezeigt.

end Testbarkeitsmetriken berechnen.

A.2.2 Anwendungsfall „Testbarkeitsmetriken erstmalig berechnen“

use case Testbarkeitsmetriken erstmalig berechnen

actors Together-Benutzer

precondition

Wie Anwendungsfall „Testbarkeitsmetriken berechnen“.

main flow

Bei der erstmaligen Berechnung der Metriken werden dem Benutzer eingangs die für die Analyse verfügbaren Testbarkeitsmetriken zur Auswahl angeboten (includes „Metriken auswählen“). Er wählt die gewünschten Metriken aus und startet die Testbarkeitsanalyse (siehe allgemeiner Anwendungsfall „Testbarkeitsmetriken berechnen“).

postcondition

Wie Anwendungsfall „Testbarkeitsmetriken berechnen“.

end Testbarkeitsmetriken erstmalig berechnen.

A.2.3 Anwendungsfall „Metriken auswählen“

use case Metriken auswählen
actors Together-Benutzer
precondition
-
main flow
Der Benutzer erhält alle für die Analyse verfügbaren Testbarkeitsmetriken zur Auswahl angeboten. Zu jeder der Metriken kann er eine kurze Beschreibung abrufen. Er wählt die gewünschten Metriken aus und startet die Testbarkeitsanalyse.
exceptional flow Keine Metriken ausgewählt
Der Benutzer erhält einen Hinweis, dass wenigstens eine Metrik zur Berechnung ausgewählt werden muss.
postcondition
Wenigstens eine der Testbarkeitsmetriken wurde ausgewählt.
end Metriken auswählen.

A.2.4 Anwendungsfall „Suchoptionen einstellen“

use case Suchoptionen einstellen
actors Together-Benutzer
precondition
-
main flow
Der Benutzer kann aus dem zu analysierenden Projekt einzelne Programmkomponenten (Pakete, Klassen) abwählen, die bei der Suche nach Abhängigkeiten weder durchlaufen noch als Ziel von Abhängigkeiten berücksichtigt werden sollen. Er kann ferner Komponenten aus dem Klassenpfad seines Projektes auswählen, die als zusätzliche Ziele von Abhängigkeiten erkannt werden sollen.
postcondition
-
end Suchoptionen einstellen.

A.2.5 Anwendungsfall „Navigation zum Sourcecode“

use case Navigation zum Sourcecode
actors Together-Benutzer
precondition
Eine Testbarkeitsanalyse wurde durchgeführt. Der Benutzer befindet sich in der Anzeige der Abhängigkeits- oder Komponentenmetriken.
main flow
Der Benutzer lässt sich den Teil des Sourcecodes anzeigen, der Auslöser für eine Abhängigkeit ist. Hierzu wird die Klasse mit der Abhängigkeit im Texteditor von Together geöffnet und die betreffende Stelle hervorgehoben. Analog können auch alle in den Metriktabellen aufgelisteten Komponenten im Texteditor von Together geöffnet werden.
postcondition
Die ausgewählte Abhängigkeit oder Komponente ist im Texteditor von Together geöffnet. Wurde zum Ursprung einer Abhängigkeit navigiert, so ist die entsprechende Textstelle hervorgehoben.
end Navigation zum Sourcecode.

A.2.6 Anwendungsfall „Navigation zu Klassendiagramm“

use case Navigation zu Klassendiagramm
actors Together-Benutzer
precondition
Eine Testbarkeitsanalyse wurde durchgeführt. Der Benutzer befindet sich in der Anzeige der Abhängigkeits- oder Komponentenmetriken.
main flow
Der Benutzer lässt sich das Klassendiagramm zu einer Komponente anzeigen. Hierzu wird das Klassendiagramm im Designfenster von Together geöffnet und die ausgewählte Komponente im Diagramm selektiert.
postcondition
Das Klassendiagramm, dessen Bestandteil die ausgewählte Komponente ist, wird im Designfenster von Together angezeigt.
end Navigation zu Klassendiagramm.

A.2.7 Anwendungsfall „Metriken speichern“

use case Metriken speichern
actors Together-Benutzer
precondition
Eine Testbarkeitsanalyse wurde durchgeführt. Der Benutzer befindet sich in der Anzeige der Abhängigkeits- oder Komponentenmetriken.
main flow
Der Benutzer speichert die angezeigten Metriken in einem Textformat zur weiteren Verarbeitung. Der Speicherort ist frei wählbar.
postcondition
Die Metriken wurde in einer Textdatei gespeichert.
end Metriken speichern.

A.2.8 Anwendungsfall „Metriken drucken“

use case Metriken drucken
actors Together-Benutzer, Drucker
precondition
Eine Testbarkeitsanalyse wurde durchgeführt. Der Benutzer befindet sich in der Anzeige der Abhängigkeits- oder Komponentenmetriken.
main flow
Der Benutzer wählt die Druckeinstellungen und druckt die angezeigten Abhängigkeits- oder Komponentenmetriken aus.
postcondition
Die Metriken wurden ausgedruckt.
end Metriken drucken.

A.2.9 Anwendungsfall „Abhängigkeitsgraph anzeigen“

use case Abhängigkeitsgraph anzeigen
actors Together-Benutzer
precondition
Die Berechnung der Testbarkeitsmetriken wurde erfolgreich ausgeführt.
main flow
Der Benutzer lässt sich die Komponenten des analysierten Projektes und die zwischen ihnen ermittelten Abhängigkeiten in graphischer Form darstellen. Die ermittelten Abhängigkeiten werden als gerichtete Kanten eines geometrischen Graphen, die abhängigen Komponenten als dessen Knoten angezeigt. Die Knoten können gemeinsam mit ihren Kanten durch Verschieben beliebig angeordnet werden. Einzelne Knoten und ihre ein- und ausgehenden Kanten können ausgeblendet werden.
alternate flow Anzeige aktualisieren
Alle Elemente des Graphen werden unter Berücksichtigung der aktuellen Größe des Anzeigefensters automatisch neu angeordnet.
alternate flow Farbschema verändern
Um bestimmte Aspekte wie Zyklen oder die Art der Abhängigkeit graphisch hervorzuheben, können die Elemente des Graphen in verschiedenartigen Färbungen ausgegeben werden.
alternate flow Speichern des Graphen
Der angezeigte Graph kann in einem Grafikformat (JPEG) gespeichert werden. Für die zugrunde liegende Struktur ist die Speicherung in Textform (XML) möglich (extension point „Graph speichern“).
alternate flow Ausdruck des Graphen
Der Benutzer startet den Ausdruck der aktuellen Anzeige des Graphen (extension point „Graph drucken“).
postcondition
-
end Abhängigkeitsgraph anzeigen.

A.2.10 Anwendungsfall „Graph speichern“

use case Graph speichern
actors Together-Benutzer
precondition
Die Anzeige des Abhängigkeitsgraphen ist sichtbar.
main flow
Der Benutzer speichert die aktuelle Anzeige des Abhängigkeitsgraphen in einem Grafikormat (JPEG). Alternativ kann auch die Struktur des Abhängigkeitsgraphen in Textform (XML-Format) gespeichert werden. Der Speicherort ist vom Benutzer jeweils frei wählbar.
postcondition
Der angezeigte Graph wurde im gewünschten Format gespeichert.
end Graph speichern.

A.2.11 Anwendungsfall „Graph drucken“

use case Graph drucken
actors Together-Benutzer, Drucker
precondition
Die Anzeige des Abhängigkeitsgraphen ist sichtbar.
main flow
Der Benutzer wählt die Druckeinstellungen und druckt den aktuell angezeigten Graph aus.
postcondition
Die aktuelle Anzeige wurde ausgedruckt.
end Graph drucken.

Anhang B Übersicht möglicher syntaktischer Abhängigkeiten in Java-Programmen

EINLEITUNG **141**

JAVA-ABHÄNGIGKEITEN **143**

B.1 VERERBUNG (SUBCLASSING) 143
B.2 ERWEITERUNG EINES SCHNITTSTELLENTYPS 143
B.3 IMPLEMENTIERUNG EINES SCHNITTSTELLENTYPS 144
B.4 TYP EINES METHODEN-PARAMETERS 145
B.5 TYP DES RÜCKGABEWERTS EINER METHODE 145
B.6 TYPSPEZIFIKATION IN DER THROWS-KLAUSEL EINER METHODE 146
B.7 TYP EINES PARAMETERS IN EINEM CATCH-BLOCK 146
B.8 TYP EINES ATTRIBUTS EINER KLASSE 146
B.9 TYP EINER LOKALEN VARIABLEN 147
B.10 LESENDER ZUGRIFF AUF NICHT-STATISCHES KLASSENATTRIBUT 148
B.11 LESENDER ZUGRIFF AUF STATISCHES KLASSENATTRIBUT 148
B.12 SCHREIBENDER ZUGRIFF AUF NICHT-STATISCHES KLASSENATTRIBUT 149
B.13 SCHREIBENDER ZUGRIFF AUF STATISCHES KLASSENATTRIBUT 149
B.14 AUFRUF EINER KLASSENFREMDEN STATISCHEN METHODE 150
B.15 AUFRUF EINER KLASSENFREMDEN NICHT-STATISCHEN METHODE 150
B.16 AUFRUF DES SUPERKLASSENKONSTRUKTORS (EXPLIZIT) 150
B.17 AUFRUF VON ÜBERSCHRIEBENEN ODER VERDECKTEN METHODEN 151
B.18 AUFRUF EINER INTERFACE-METHODE 152
B.19 KLASSENINTERNER AUFRUF EINER NICHT-STATISCHEN METHODE 152
B.20 KLASSENINTERNER AUFRUF EINER STATISCHEN METHODE 152
B.21 INSTANZIERUNG MITTELS NEW-OPERATOR 153
B.22 INITIALISIERUNG EINES EINDIMENSIONALEN ARRAYS 154
B.23 INITIALISIERUNG EINES MEHRDIMENSIONALEN ARRAYS 154
B.24 VERWENDUNG DES TYPE-CAST-OPERATORS 154
B.25 VERWENDUNG DES INSTANCEOF-OPERATORS 155

JAVA-ABHÄNGIGKEITEN IN BESONDEREM KONTEXT **156**

B.26 LOKALE INNERE KLASSEN (STATISCH UND NICHT-STATISCH) 156
B.27 ANONYME INNERE KLASSEN 156
B.28 STATISCHER KONSTRUKTOR (INITIALISIERER) 157
B.29 INSTANZ-INITIALISIERER 158
B.30 ABHÄNGIGKEITEN ZU VERERBTEN METHODEN UND ATTRIBUTEN 159
B.31 MANUELLE TOGETHER-ABHÄNGIGKEITEN 159

EINSCHRÄNKUNGEN **161**

ZUSAMMENFASSUNG **162**

Einleitung

Für die Erstellung dieses Dokuments wurden die Syntax und die Struktur von Java-Programmen analysiert, um Aufschluss über die Abhängigkeiten zu erhalten, die in Java-Programmen auftreten können. Die Ergebnisse dieser Analyse werden in den nachfolgenden Abschnitten in Form der gefundenen Abhängigkeiten aufgelistet.

Das Konzept der Together-Erweiterung zur Testbarkeitsanalyse sieht vor, diese Abhängigkeiten in einer Graphstruktur zu speichern und sie zur Berechung von Testbarkeitsmetriken in Form dieses Graphen an *ImproveT* weiter zu reichen. In den nachfolgenden Abschnitten wird deshalb auch für jede Abhängigkeit beschrieben, in welcher Form sie ihre Repräsentation im Graph findet.

Zum besseren Verständnis der beschriebenen Grapheinträge wird in nachstehender Abbildung versucht, anhand einiger weniger Beispieleinträge bereits vorab die Struktur des entstehenden Abhängigkeitsgraphen zu veranschaulichen. Man erkennt die Ebenenstruktur des Graphen, die Darstellung der Komponenten des Systems als Knoten und der Abhängigkeiten als Kanten des Graphen und die hierarchische Verknüpfung der Knoten und Kanten der jeweils benachbarten Ebenen:

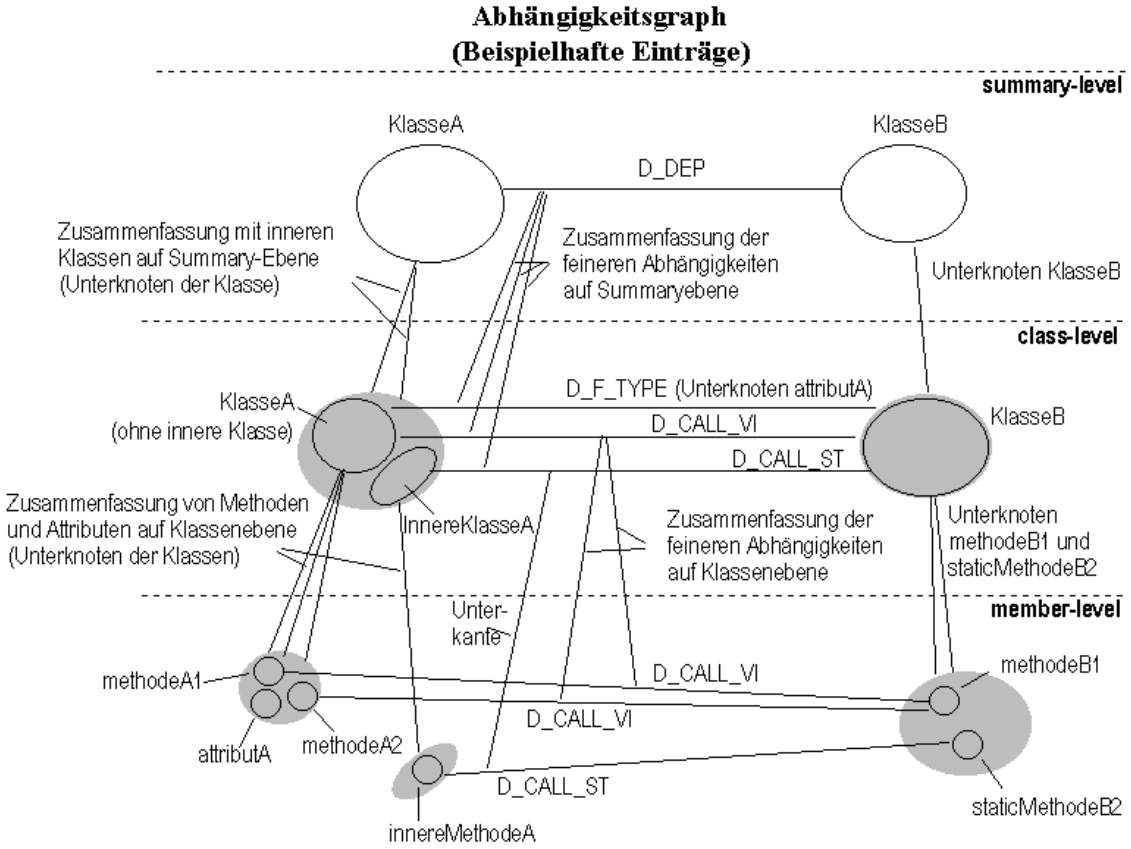


Abbildung 1: Beispielhafte Struktur eines Abhängigkeitsgraphen

In der Abbildung sind bereits einige Abhängigkeitstypen zu erkennen. Die für die verschiedenen Typen verwendeten Kürzel sind durch ImproveT vorgegeben. Die Klassifizierung in verschiedene Typen erfolgt nicht nur für Abhängigkeiten sondern auch für die im Graph als Knoten gespeicherten Komponenten. Folgende Komponenten und ihre in ImproveT festgelegten Kürzel treten in den

nachfolgenden Abschnitten auf, wobei die feinere Klassifizierung immer Vorrang vor der allgemeineren Beschreibung haben wird:

- Klassen (T_CLASS)
- Finale Klassen (T_CL_FINAL)
- Interfaces (T_INTERF)
- Abstrakte Klassen (T_ABSTR)
- Innere Klassen (T_INRCL)
- Konstruktoren (T_CONSTR)
- Methoden (T_METHOD)
- Finale Methoden (T_M_FINAL)
- Attribute (T_FIELD)
- Finale Attribute (T_F_FINAL)
- Initialisierer (T_INIT).

Eine zusammenfassende Auflistung aller in den nachfolgenden Abschnitten definierten Abhängigkeitstypen sowie ein Überblick über die Möglichkeit des Auftretens der Abhängigkeiten und Komponenten in den verschiedenen Ebenen des Abhängigkeitsgraphen befindet sich am Ende dieses Dokuments.

Java-Abhängigkeiten

B.1 Vererbung (Subclassing)

B.1.1 Erläuterung und Beispiel:

Hierunter fallen Abhängigkeiten, die durch Vererbung von Programmteilen durch explizite Angabe des Schlüsselworts `extends` definiert werden: `Subclass` **`extends`** `Superclass`. Bei Subklasse kann es sich insbesondere auch um abstrakte oder finale Klassen handeln.

```
public class Firma extends Geschaeftspartner {
   ...
}
```

Folgt bei der Definition einer anonymen Klasse hinter dem Schlüsselwort `new` der Name einer Klasse, so handelt es sich auch ohne Angabe des Schlüsselworts `extends` bei der anonymen Klasse um eine Subklasse dieser Klasse:

```
public class Firma extends Geschaeftspartner {
   ...
   Date datum = new Date() { ...};
}
```

B.1.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Die Grapheinträge bei der expliziten Vererbung mittels `extends` lauten wie folgt:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Firma	T_CLASS	Geschaeftspartner	T_CLASS
Class	D_EXTEND	Firma	T_CLASS	Geschaeftspartner	T_CLASS
Member	-	-	-	-	-

Bei anonymen Klassen verläuft die Abhängigkeit auf Klassenebene von der anonymen Klasse zur Superklasse. Auf Summary-Klassenebene hingegen geht die Abhängigkeit von der Klasse aus, in der die anonyme Klasse definiert wird:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Firma	T_CLASS	Date	T_CLASS
Class	D_EXTEND	anonym_class	T_INRCL	Date	T_CLASS
Member	-	-	-	-	-

Anmerkung: Abstrakte, finale sowie innere und anonyme Klassen werden durch eigene Typen (T_ABSTR, T_CL_FINAL, T_INRCL) repräsentiert. Dies gilt auch für alle nachstehend beschriebenen Abhängigkeiten, sofern eine Komponente dieses Typs daran beteiligt sein kann.

B.2 Erweiterung eines Schnittstellentyps

B.2.1 Erläuterung und Beispiel:

Die Erweiterung eines Schnittstellentyps (Interface) erfolgt – wie bei der Vererbung – durch das Schlüsselwort `extends`:

```
public interface IBelegung extends IErweitertPersistent {
   ...
}
```

B.2.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	IBelegung	T_INTERF	IErweiterPersistent	T_INTERF
Class	D_EXTEND	IBelegung	T_INTERF	IErweiterPersistent	T_INTERF
Member	-	-	-	-	-

B.3 Implementierung eines Schnittstellentyps

B.3.1 Erläuterung und Beispiel:

Möchte eine Klasse ein Interface als Typ implementieren, so folgt hinter dem Klassennamen das Schlüsselwort `implements` gefolgt vom Schnittstellentyp. Die implementierende Klasse kann auch als abstrakt definiert sein:

```
public abstract class Firma implements IFirma {
   ...
}
```

Steht im Rahmen der Definition einer anonymen Klasse hinter dem Schlüsselwort new der Name eines Schnittstellentyps, so implementiert die anonyme Klasse dieses Interface. Es wird ebenfalls eine entsprechende Abhängigkeit modelliert:

```
public class Firma extends Geschaeftspartner {
   ...
   ActionListener a = new ActionListener() { ...};
}
```

B.3.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Die Grapheinträge bei expliziter Implementierung eines Interfaces lauten wie folgt:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Firma	T_ABSTR	IFirma	T_INTERF
Class	D_IMPLEM	Firma	T_ABSTR	IFirma	T_INTERF
Member	-	-	-	-	-

Bei anonymen Klassen verläuft die Abhängigkeit auf Klassenebene von der anonymen Klasse zur Superklasse. Auf Summary-Klassenebene hingegen geht die Abhängigkeit von der Klasse aus, in der die anonyme Klasse definiert wird:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Firma	T_CLASS	ActionListener	T_CLASS
Class	D_IMPLEM	anonym_class	T_INRCL	ActionListener	T_CLASS

Abstrakte Klassen, innere und anonyme Klassen werden durch eigene Typen (T_ABSTR, T_INCRL) repräsentiert.

B.4 Typ eines Methoden-Parameters

B.4.1 Erläuterung und Beispiel:

Besitzt eine Methode als Typ eines ihrer Parameter einen Objekttyp, so wird eine Abhängigkeit vom Typ D_PARAM definiert. Zu den Objekttypen zählen beispielsweise auch die mit der Java™ 2 Platform Standard Edition (J2SE) ausgelieferten Wrapper-Klassen (`Integer`, `Boolean`, ...) oder die Klasse `String`. Die Java-Basisdatentypen (`int`, `boolean`, ...) bleiben hingegen unberücksichtigt.

```
public class Person extends Geschaeftspartner implements IPerson {
    ...
   public void loeseAnstellungFirma(IFirma f) throws DatenbankAusnahme {
      AnstellungRelation.getEinzigeInstanz().loese((IPerson)getThis(), f);
   }
}
```

B.4.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Person	T_CLASS	IFirma	T_INTERF
Class	D_PARAM	Person	T_CLASS	IFirma	T_INTERF
Member	-	-	-	-	-

Um den Bezug zur abhängigkeitsverursachenden Methode nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Methodenknoten gespeichert.

B.5 Typ des Rückgabewerts einer Methode

B.5.1 Erläuterung und Beispiel:

Liefert eine Methode als Returnwert einen Objekttyp zurück, so wird eine Abhängigkeit vom Typ D_RETURN definiert. Zu den Objekttypen zählen beispielsweise auch die mit der Java™ 2 Platform Standard Edition (J2SE) ausgelieferten Wrapper-Klassen (`Integer`, `Boolean`, ...) oder die Klasse `String`. Die Java-Basisdatentypen (`int`, `boolean`, ...) bleiben hingegen unberücksichtigt.

```
public class Person extends Geschaeftspartner implements IPerson {
   ...
   public IFirma getAnstellungFirma() throws DatenbankAusnahme {
      return AnstellungRelation.getEinzigeInstanz().getFirma((IPerson)getThis());
   }
   ...
}
```

B.5.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Person	T_CLASS	IFirma	T_INTERF
Class	D_RETURN	Person	T_CLASS	IFirma	T_INTERF
Member	-	-	-	-	-

Um den Bezug zur abhängigkeitsverursachenden Methode nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Methodenknoten gespeichert.

B.6 Typspezifikation in der throws-Klausel einer Methode

B.6.1 Erläuterung und Beispiel:

Zeigt eine Methode durch Angabe einer Ausnahme in ihrer `throws`-Klausel an, dass sie eine bestimmte Exception nicht selbst behandelt, sondern diese an die aufrufende Methode weitergibt, so wird hierdurch eine Abhängigkeit vom Typ D_THROWS definiert:

```
public class Person extends Geschaeftspartner implements IPerson {
   public static IPerson getObjekt(String einSchluessel)
      throws UngueltigerSchluesselException, DatenbankAusnahme {
      ...
   }
}
```

B.6.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Person	T_CLASS	UngueltigerSchluesselException	T_CLASS
Class	D_THROWS	Person	T_CLASS	UngueltigerSchluesselException	T_CLASS
Member	-	-	-	-	-

Um den Bezug zur abhängigkeitsverursachenden Methode nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Methodenknoten gespeichert.

B.7 Typ eines Parameters in einem catch-Block

B.7.1 Erläuterung und Beispiel:

Wird innerhalb eines überwachten Programmteiles durch Verwendung einer `catch`-Klausel ein Fehler abgefangen und behandelt, so wird eine Abhängigkeit vom Typ D_CATCH zur behandelten Ausnahme definiert:

```
public class SeminarisDatenbank extends Datenbank {
   public static SeminarisDatenbank getEinzigeInstanz() {
      try {
         ...
      } catch( KonfigurationException ke) {
         ...
      }
   }
}
```

B.7.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	SeminarisDatenbank	T_CLASS	KonfigurationException	T_CLASS
Class	D_CATCH	SeminarisDatenbank	T_CLASS	KonfigurationException	T_CLASS
Member	-	-	-	-	-

Um den Bezug zur abhängigkeitsverursachenden Methode nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Methodenknoten gespeichert.

B.8 Typ eines Attributs einer Klasse

B.8.1 Erläuterung und Beispiel:

Definiert eine Klasse ein Attribut vom Typ eines Objektes, so wird eine Abhängigkeit vom Typ D_F_TYPE definiert. Zu den Objekttypen zählen beispielsweise auch die mit der Java™ 2 Platform

Standard Edition (J2SE) ausgelieferten Wrapper-Klassen (`Integer`, `Boolean`, ...) oder die Klasse `String`. Die Java-Basisdatentypen (`int`, `boolean`, ...) bleiben hingegen unberücksichtigt.

```
public class SeminarisDatenbank extends Datenbank {
   ...
   private ProtokollManager protokollManager;
   ...
}
```

B.8.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	SeminarisDatenbank	T_CLASS	ProtokollManager	T_CLASS
Class	D_F_TYPE	SeminarisDatenbank	T_CLASS	ProtokollManager	T_CLASS
Member	-	-	-	-	-

Um den Bezug zum abhängigkeitsverursachenden Attribut nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Attributknoten gespeichert.

Es erfolgt bezüglich des Typs keine Unterscheidung zwischen statischen und nicht-statischen Klassenattributen.

Attribute vom eigenen Klassentyp (z.B. Singleton-Instances) werden nicht als Abhängigkeit erkannt.

B.9 Typ einer lokalen Variablen

B.9.1 Erläuterung und Beispiel:

Innerhalb eines Methodenrumpfes können lokale Variablen definiert werden. Handelt es sich beim Typ der lokalen Variablen um einen Objekttyp, so wird eine Abhängigkeit vom Typ `D_V_TYPE` festgestellt. Zu den Objekttypen zählen beispielsweise auch die mit der Java™ 2 Platform Standard Edition (J2SE) ausgelieferten Wrapper-Klassen (`Integer`, `Boolean`, ...) oder die Klasse `String`. Die Java-Basisdatentypen (`int`, `boolean`, ...) bleiben hingegen unberücksichtigt:

```
public class FirmaLoeschenK {
   public void loeschen(String firmaSchluessel) throws ... {
      IFirma persistenteFirma = Firma.getObjekt(firmaSchluessel);
      ...
   }
}
```

Lokale Variablen vom eigenen Klassentyp werden nicht als Abhängigkeit erkannt.

B.9.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaLoeschenK	T_CLASS	IFirma	T_INTERF
Class	D_V_TYPE	FirmaLoeschenK	T_CLASS	IFirma	T_INTERF
Member	-	-	-	-	-

Um den Bezug zur abhängigkeitsverursachenden Methode nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Methodenknoten gespeichert.

B.10 Lesender Zugriff auf nicht-statisches Klassenattribut

B.10.1 Erläuterung und Beispiel:

Wird lesend auf den Wert eines Attributes einer Klasse zugegriffen, so wird hierdurch eine Abhängigkeit vom Typ D_USES definiert:

```
abstract public class FirmaAendernErfassenK {
   protected Firma transienteFirma;
   ...
}
```

```
public class FirmaErfassenK extends FirmaAendernErfassenK {
   public Firma getObjektFirma() {
      ...
      return transienteFirma;
   }
}
```

B.10.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaErfassenK	T_CLASS	FirmaAendernErfassenK	T_ABSTR
Class	D_USES	FirmaErfassenK	T_CLASS	FirmaAendernErfassenK	T_ABSTR
Member	D_USES	getObjektFirma()	T_METHOD	transienteFirma	T_FIELD

Wird auf ein Attribut innerhalb der eigenen Klasse zugegriffen, so wird ebenfalls eine Abhängigkeit auf Memberebene definiert. Auf Klassenebene bleiben klasseninterne Abhängigkeiten jedoch unberücksichtigt.

Der Zugriff auf lokale Methodenvariablen bleibt auf allen Ebenen unberücksichtigt.

B.11 Lesender Zugriff auf statisches Klassenattribut

B.11.1 Erläuterung und Beispiel:

Wird lesend auf den Wert eines statischen Attributes einer Klasse zugegriffen, so wird hierdurch eine Abhängigkeit vom Typ D_USES_ST definiert:

```
abstract public class FirmaAendernErfassenK {
   ...
   public SeminarisMeldung speichern() throws DatenbankAusnahme {
        ...
        SeminarisK.Database.starteTransaktion();
        ...
   }
   ...
}
```

B.11.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaAendernErfassenK	T_ABSTR	SeminarisK	T_CLASS
Class	D_USES_ST	FirmaAendernErfassenK	T_ABSTR	SeminarisK	T_CLASS
Member	D_USES_ST	speichern()	T_METHOD	Database	T_FIELD

Wird auf ein statisches Attribut innerhalb der eigenen Klasse zugegriffen, so wird ebenfalls eine Abhängigkeit auf Memberebene definiert. Auf Klassenebene bleiben klasseninterne Abhängigkeiten jedoch unberücksichtigt.

B.12 Schreibender Zugriff auf nicht-statisches Klassenattribut

B.12.1 Erläuterung und Beispiel:

Wird der Wert eines Attributes einer Klasse neu gesetzt, so wird hierdurch eine Abhängigkeit vom Typ D_DEF definiert:

```
abstract public class FirmaAendernErfassenK {
   protected Firma transienteFirma;
   ...
}
```

```
public class FirmaErfassenK extends FirmaAendernErfassenK {
   public Firma getObjektFirma() {
      ...
      transienteFirma = Firma.erzeugeTransient();
   }
}
```

B.12.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaErfassenK	T_CLASS	FirmaAendernErfassenK	T_ABSTR
Class	D_DEF	FirmaErfassenK	T_CLASS	FirmaAendernErfassenK	T_ABSTR
Member	D_DEF	getObjektFirma()	T_METHOD	transienteFirma	T_FIELD

Wird auf ein Attribut innerhalb der eigenen Klasse zugegriffen, so wird ebenfalls eine Abhängigkeit auf Memberebene definiert. Auf Klassenebene bleiben klasseninterne Abhängigkeiten jedoch unberücksichtigt.

Der Zugriff auf lokale Methodenvariablen bleibt auf allen Ebenen unberücksichtigt.

B.13 Schreibender Zugriff auf statisches Klassenattribut

B.13.1 Erläuterung und Beispiel:

Wird der Wert eines statischen Attributes einer Klasse neu gesetzt, so wird hierdurch eine Abhängigkeit vom Typ D_DEF_ST definiert:

```
public class SeminarisK {
   private static int nextKundennummer;
   ...
}
```

```
public class FirmaErfassenK {
   public Firma getObjektFirma() {
      ...
      nextKundennummer = 12345;
   }
}
```

B.13.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaErfassenK	T_CLASS	SeminarisK	T_CLASS
Class	D_DEF_ST	FirmaErfassenK	T_CLASS	SeminarisK	T_CLASS
Member	D_DEF_ST	getObjektFirma()	T_METHOD	nextKundennummer	T_FIELD

Wird auf ein Attribut innerhalb der eigenen Klasse zugegriffen, so wird ebenfalls eine Abhängigkeit auf Memberebene definiert. Auf Klassenbebene bleiben klasseninterne Abhängigkeiten jedoch unberücksichtigt.

B.14 Aufruf einer klassenfremden statischen Methode

B.14.1 Erläuterung und Beispiel:

Wird eine als statisch definierte Methode einer fremden Klasse aufgerufen, so wird hierdurch eine Abhängigkeit vom Typ D_CALL_ST definiert:

```
abstract public class Seminarveranstaltung ... {
   public static Iterator getAlle() throws DatenbankAusnahme {
      return SeminarveranstaltungOrdner.getEinzigeInstanz().getSVs();
   }
}
```

Als fremde Klasse gelten auch alle äußeren Klassen einer inneren Klasse. Der Aufruf einer klasseneigenen statischen Methode wird als Abhängigkeit D_CALL_TS gespeichert.

B.14.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Seminarveranstaltung	T_ABSTR	Seminarveranstaltung Ordner	T_CLASS
Class	D_CALL_ST	Seminarveranstaltung	T_ABSTR	Seminarveranstaltung Ordner	T_CLASS
Member	D_CALL_ST	getAlle()	T_METHOD	getEinzigeInstanz()	T_METHOD

B.15 Aufruf einer klassenfremden nicht-statischen Methode

B.15.1 Erläuterung und Beispiel:

Wird eine nicht-statische Methode einer fremden Klasse aufgerufen, so wird hierdurch eine Abhängigkeit vom Typ D_CALL_VI definiert:

```
abstract public class Seminarveranstaltung ... {
   public static Iterator getAlle() throws DatenbankAusnahme {
      return SeminarveranstaltungOrdner.getEinzigeInstanz().getSVs();
   }
}
```

Als fremde Klasse gelten auch alle äußeren Klassen einer inneren Klasse. Der Aufruf einer klasseneigenen nicht-statischen Methode wird als Abhängigkeit D_CALL_TI gespeichert.

B.15.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Seminarveranstaltung	T_ABSTR	Seminarveranstaltung Ordner	T_CLASS
Class	D_CALL_VI	Seminarveranstaltung	T_ABSTR	Seminarveranstaltung Ordner	T_CLASS
Member	D_CALL_VI	getAlle()	T_METHOD	getSVs()	T_METHOD

B.16 Aufruf des Superklassenkonstruktors (explizit)

B.16.1 Erläuterung und Beispiel:

Ruft ein Konstruktor einen der Konstruktoren der Superklasse mit der Anweisung `super(...)` auf, so wird eine Abhängigkeit D_CALL_SU definiert. Wird in abgeleiteten Klassen auf den Aufruf des Konstruktors der Superklasse verzichtet, so fügt der Compiler während des Übersetzungsvorgangs als

erste Anweisung einen implizit generierten `super()`-Aufruf in den Konstruktor ein. Dieser vom Compiler eingefügte Aufruf des Superklassenkonstruktors bleibt bei der Ermittlung dieses Abhängigkeitstyps unberücksichtigt.

```
public class FirmaErfassenK extends FirmaAendernErfassenK {

    protected FirmaErfassenK() {
        super();
    }
    ...
}
```

Ruft ein Konstruktor mittels der `this`-Anweisung einen anderen Konstruktor derselben Klasse auf, wird dies als klasseninterne Abhängigkeit zu einer statischen Methode gespeichert (D_CALL_TS).

B.16.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Der explizite Aufruf eines Superklassenkonstruktors über die `super()`-Anweisung führt zu folgenden Einträgen im Abhängigkeitsgraph:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaErfassenK	T_CLASS	FirmaAendernErfassenK	T_CLASS
Class	D_CALL_SU	FirmaErfassenK	T_CLASS	FirmaAendernErfassenK	T_CLASS
Member	D_CALL_SU	constructor()	T_CONSTR	constructor()	T_CONSTR

B.17 Aufruf von überschriebenen oder verdeckten Methoden

B.17.1 Erläuterung und Beispiel:

Innerhalb einer Subklasse kann auf überschriebene oder verdeckte Methoden von Superklassen mittels des Schlüsselwortes `super` zugegriffen werden. Außer dem Namen hat dies mit dem Aufruf des Superklassen-Konstruktors nichts gemeinsam:

```
public class Firma extends Geschaeftspartner ... {
   public void setEqual( IFirma eineFirma) {
      super.setEqual( eineFirma);
      ...
   }
}
```

In diesen Fällen kann bereits im Rahmen der Kompilierung ermittelt werden, welche Methode aufgerufen wird (die überschriebene Methode der Superklasse, die in der Hierarchie am nächsten liegt). Somit kann der Methodenaufruf vom Compiler statisch gebunden werden. Dies kann ferner unabhängig vom Typ der aufgerufenen Methode (statisch oder nicht-statisch) erfolgen. Folglich liegt eine Abhängigkeit D_CALL_ST vor.

B.17.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

In obigem Beispiel wird zwar eine nicht-statische Methode aufgerufen, der Eintrag in den Abhängigkeitsgraphen erfolgt dennoch mit Typ D_CALL_ST:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Firma	T_CLASS	Geschaeftspartner	T_CLASS
Class	D_CALL_ST	Firma	T_CLASS	Geschaeftspartner	T_CLASS
Member	D_CALL_ST	setEqual(...)	T_METHOD	setEqual(...)	T_METHOD

B.18 Aufruf einer Interface-Methode

B.18.1 Erläuterung und Beispiel:

Wird eine in einem Schnittstellentyp (Interface) definierte Methode aufgerufen, so wird hierdurch eine Abhängigkeit vom Typ D_CALL_IF definiert:

```
abstract public class Geschaeftspartner implements IGeschaeftspartner {
  public static Collection getAlleSchluessel() throws DatenbankAusnahme {
     ...
     while (iter.hasNext()) {
       ergebnisMenge.add(((IGeschaeftspartner)iter.next()).getSchluessel());
     }
  }
}
```

B.18.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Geschaeftspartner	T_ABSTR	IGeschaeftspartner	T_INTERF
Class	D_CALL_IF	Geschaeftspartner	T_ABSTR	IGeschaeftspartner	T_INTERF
Member	D_CALL_IF	getAlleSchluessel()	T_METHOD	getSchluessel()	T_METHOD

B.19 Klasseninterner Aufruf einer nicht-statischen Methode

B.19.1 Erläuterung und Beispiel:

Wird eine nicht-statische Methode derselben Klasse aufgerufen, so wird hierdurch eine Abhängigkeit vom Typ D_CALL_TI definiert. Der Aufruf einer Methode einer äußeren Klasse aus einer inneren Klasse wird nicht als klasseninterner Aufruf betrachtet, sondern als Abhängigkeit D_CALL_VI gespeichert.

```
public class Firma ... {
   public String getName() {
       return name;
   }
   ...
   public String getAnrede() {
       return ... + getName() + ...;
   }
}
```

B.19.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	-	-	-	-	-
Class	-	-	-	-	-
Member	D_CALL_TI	getAnrede()	T_METHOD	getName()	T_METHOD

Klasseninterne Abhängigkeiten werden nur auf Memberebene gespeichert.

B.20 Klasseninterner Aufruf einer statischen Methode

B.20.1 Erläuterung und Beispiel:

Wird eine statische Methode derselben Klasse aufgerufen, so wird hierdurch eine Abhängigkeit vom Typ D_CALL_TS definiert. Der Aufruf einer statischen Methode einer äußeren Klasse aus einer inneren oder anonymen Klasse wird nicht als klasseninterner Aufruf betrachtet, sondern als Abhängigkeit D_CALL_ST gespeichert.

```
public class Firma ... {
   public static String getClassName() {
       return "Firma";
   }
   ...
   public void protocol( String meldung) {
       return getClassName() + ":" + meldung;
   }
}
```

Ruft ein Konstruktor mittels der this-Anweisung einen anderen Konstruktor derselben Klasse auf, so wird dies ebenfalls als klasseninterne Abhängigkeit zu einer statischen Methode (Typ T_CONSTR) gespeichert.

B.20.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	-	-	-	-	-
Class	-	-	-	-	-
Member	D_CALL_TS	protocol()	T_METHOD	getClassName()	T_METHOD

Klasseninterne Abhängigkeiten werden nur auf Memberebene gespeichert.

B.21 Instanzierung mittels new-Operator

B.21.1 Erläuterung und Beispiel:

Beim Erzeugen einer neuen Instanz einer Klasse mit dem Operator new wird eine Abhängigkeit D_NEW festgestellt:

```
abstract public class FirmaAendernErfassenAA extends SeminarisFrame {
   ...
   protected FirmaGesamtS firmaGesamtS = new FirmaGesamtS();      ...
   ...
}
```

B.21.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Handelt es sich bei dem zur Erzeugung genutzten Konstruktor um den nicht implementierten Default-Konstruktor einer Klasse, so wird die Abhängigkeit zur Klasse der neuen Instanz modelliert. Um nicht den Bezug zu dem Attribut oder der Methode, aus der der Aufruf erfolgt, zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den die Abhängigkeit auslösenden Elementknoten gespeichert:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaAendernErfassenAA	T_ABSTR	FirmaGesamtS	T_CLASS
Class	D_NEW	FirmaAendernErfassenAA	T_ABSTR	FirmaGesamtS	T_CLASS
Member	-	-	-	-	-

Ist der Konstruktor implementiert, der zur Erzeugung der neuen Instanz verwendet wird, so führt die Abhängigkeit auf Memberebene zu diesem Konstruktor:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaAendernErfassenAA	T_ABSTR	FirmaGesamtS	T_CLASS
Class	D_NEW	FirmaAendernErfassenAA	T_ABSTR	FirmaGesamtS	T_CLASS
Member	D_NEW	firmaGesamtS	T_FIELD	FirmaGesamtS()	T_CONSTR

B.22 Initialisierung eines eindimensionalen Arrays

B.22.1 Erläuterung und Beispiel:

Bei der Initialisierung eines eindimensionalen Arrays mit dem new-Operator wird eine Abhängigkeit D_NEW_AR definiert:

```
public class InitArrayExample {
   ...
   public void InitArrayMethod() {
      ClassType[] var = new ClassType[5];
      ...
   }
}
```

B.22.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	InitArrayExample	T_CLASS	ClassType	T_CLASS
Class	D_NEW_AR	InitArrayExample	T_CLASS	ClassType	T_CLASS
Member	-	-	-	-	-

Um den Bezug zur Methode oder zum Attribut, das die Abhängigkeit verursacht, nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Elementknoten gespeichert.

B.23 Initialisierung eines mehrdimensionalen Arrays

B.23.1 Erläuterung und Beispiel:

Bei der Initialisierung eines mehrdimensionalen Arrays mit dem new-Operator wird eine Abhängigkeit D_NEW_MAR definiert:

```
public class InitMultiArrayExample {
   ...
   public void InitMultiArrayMethod() {
      ClassType[][] var = new ClassType[5][2];
      ...
   }
}
```

B.23.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	InitMultiArrayExample	T_CLASS	ClassType	T_CLASS
Class	D_NEW_MAR	InitMultiArrayExample	T_CLASS	ClassType	T_CLASS
Member	-	-	-	-	-

Um den Bezug zur Methode oder zum Attribut, das die Abhängigkeit verursacht, nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Elementknoten gespeichert.

B.24 Verwendung des Type-Cast-Operators

B.24.1 Erläuterung und Beispiel:

Wird mit Hilfe des Type-Cast-Operators eine explizite Typumwandlung vorgenommen, so wird eine Abhängigkeit D_CAST ermittelt:

```
abstract public class SeminarbelegungAA extends SeminarisFrame {
   protected void darstellenInhalte() {
      ...
      gpFirmaS.darstellenInhalte( (Firma)gp);
      ...
   }
}
```

B.24.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	SeminarbelegungAA	T_ABSTR	Firma	T_CLASS
Class	D_CAST	SeminarbelegungAA	T_ABSTR	Firma	T_CLASS
Member	-	-	-	-	-

Um den Bezug zur Methode oder zum Attribut, das die Abhängigkeit verursacht, nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Elementknoten gespeichert.

B.25 Verwendung des instanceof-Operators

B.25.1 Erläuterung und Beispiel:

Wird mit Hilfe des `instanceof`-Operators ein Typvergleich durchgeführt, wird eine D_INSTANC-Abhängigkeit ermittelt:

```
public class MeldungsAnzeigeA {
   private int zeige(SeminarisMeldung eineSeminarisMeldung, Component cp)  {
      if (eineSeminarisMeldung instanceof ExceptionNachricht) {
         ...
      }
   }
}
```

B.25.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	MeldungsAnzeigeA	T_CLASS	ExceptionNachricht	T_CLASS
Class	D_INSTANC	MeldungsAnzeigeA	T_CLASS	ExceptionNachricht	T_CLASS
Member	-	-	-	-	-

Um den Bezug zur Methode oder zum Attribut, das die Abhängigkeit verursacht, nicht zu verlieren, wird bei der auf Klassenebene eingefügten Abhängigkeit eine Referenz auf den Elementknoten gespeichert.

Java-Abhängigkeiten in besonderem Kontext

B.26 Lokale innere Klassen (statisch und nicht-statisch)

B.26.1 Erläuterung und Beispiel:

In nachstehendem Beispiel handelt es sich bei der Klasse `UseCaseActionListener` um eine innere Klasse der Klasse `SachbearbeiterA`. Klassen, die nicht innerhalb anderer Klassen definiert werden, werden nachfolgend als äußere Klassen bezeichnet. Bei der Klasse `SachbearbeiterA` handelt es sich um eine äußere Klasse:

```
public class SachbearbeiterA ... {
   private SachbearbeiterA mainWindow = SeminarisH.MainWindow;
   ...
   protected class UseCaseActionListener implements ActionListener {
      public void actionPerformed(ActionEvent event) {
         try {
            ...
         } catch (ClassNotFoundException e) {
             MeldungsAnzeigeA.anzeigen( "Class missing", mainWindow);
         }
         ...
      }
   }
}
```

B.26.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	SachbearbeiterA	T_CLASS	MeldungsAnzeigeA	T_CLASS
Class	D_CALL_ST	UseCaseActionListener	T_INRCL	MeldungsAnzeigeA	T_CLASS
Class	D_USES_ST	UseCaseActionListener	T_INRCL	SachbearbeiterA	T_CLASS
Member	D_CALL_ST	actionPerformed(...)	T_METHOD	anzeigen(...)	T_METHOD
Member	D_USES_ST	actionPerformed(...)	T_METHOD	MainWindow	T_FIELD

Innere Klassen werden auf Member- und Klassenebene wie eine eigenständige Klasse behandelt. Auf Summary-Ebene wird die innere Klasse als Bestandteil der äußersten Klasse betrachtet, so dass

- Abhängigkeiten von der inneren zu ihrer äußeren Klasse als klasseninterne Abhängigkeit auf Summary-Ebene verschwinden und
- alle anderen Abhängigkeiten von der äußeren Klasse ausgehend modelliert werden.

B.27 Anonyme innere Klassen

Anonyme innere Klassen werden völlig analog zu benamten inneren Klassen behandelt. Die einzige Besonderheit ist, dass für anonyme Klassen ein eindeutiger Name generiert werden muss. So wird beispielsweise eine anonyme Klasse der Klasse SachbearbeiterA mit "...SachbearbeiterA.anonym_class_%" bezeichnet, wobei % für die fortlaufende Nummerierung der anonymen Klassen innerhalb einer Klasse steht.

B.28 Statischer Konstruktor (Initialisierer)

B.28.1 Erläuterung und Beispiel:

Sofern möglich, sollte die Initialisierung der statischen Attribute sofort bei der Deklaration erfolgen. Wird dies bei einer komplexen Initialisierung in einem statischen Konstruktor (*static initializer*) durchgeführt, so wird dieser statische Konstruktor als Element T_INIT im Graph gespeichert:

```
class FirmaLoeschenK {
   static IFirma firma;

   static {
      // statischer Initialisierungs-Block
      firma = (IFirma)Firma.erzeuge();
   }
}
```

Ein statischer Initialisierer besitzt keine Parameter und wird nur einmal unmittelbar nach dem Laden der Klasse aufgerufen. Eine Klasse kann mehrere statische Initialisierer besitzen.

B.28.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Abhängigkeit(en) auf Memberebene

Führt die Abhängigkeit zu einer Methode oder einem Attribut, so erscheint auch auf Memberebene ein entsprechender Eintrag:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaLoeschenK	T_CLASS	Firma	T_CLASS
Class	D_CALL_ST	FirmaLoeschenK	T_CLASS	Firma	T_CLASS
Member	D_CALL_ST	static_init_1()	T_INIT	erzeuge()	T_METHOD

Initialisierer sind in Together namenlos. Initialisierer derselben Klasse sind also durch ihren Namen nicht unterscheidbar. Die Benennung eines statischen Initialisierers erfolgt durch Generierung eines Namens der Art “static_init_%”, wobei das Prozentzeichen für eine fortlaufende Nummerierung innerhalb einer Klasse steht.

Abhängigkeit(en) auf Klassenebene

Führt die Abhängigkeit zu einer Klasse, so wird die Abhängigkeit nur auf Klassen- und Summary-Ebene gespeichert:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaLoeschenK	T_CLASS	Firma	T_CLASS
Class	D_CAST	FirmaLoeschenK	T_CLASS	IFirma	T_CLASS
Member	-	-	-	-	-

Bei der auf Klassenebene eingefügten Abhängigkeit wird eine Referenz zum Knoten des die Abhängigkeit auslösenden statischen Initialisierers gespeichert.

B.29 Instanz-Initialisierer

B.29.1 Erläuterung und Beispiel:

Ein Instanz-Initialisierer (*instance initializer*) besteht nur aus geschweiften Klammern, dem Initialisierungsblock. Wird im Initialisierungsblock eine Abhängigkeit begründet, so wird der Instanz-Initialisierer als Element T_INIT im Graph gespeichert:

```
class FirmaLoeschenK {
   private IFirma firma;

   {
      // Initialisierungs-Block
      firma = (IFirma)Firma.erzeuge();
   }

   // Konstruktor
   public FirmaLoeschenK() {
      ...
   }
}
```

Ein Instanz-Initialisierer besitzt keine Parameter und wird genau einmal für jede erzeugte Instanz ausgeführt. In einer Klasse können mehrere Instanz-Initialisierer vorkommen.

B.29.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Abhängigkeit(en) auf Memberebene

Führt die Abhängigkeit zu einer Methode oder einem Attribut, so erscheint auch auf Memberebene ein entsprechender Eintrag:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaLoeschenK	T_CLASS	Firma	T_CLASS
Class	D_CALL_ST	FirmaLoeschenK	T_CLASS	Firma	T_CLASS
Member	D_CALL_ST	instance_init_1()	T_INIT	erzeuge()	T_METHOD

Initialisierer sind in Together namenlos. Initialisierer derselben Klasse sind also durch ihren Namen nicht unterscheidbar. Die Benennung eines Instanz-Initialisierers erfolgt durch Generierung eines Namens der Art “instance_init_%”, wobei das Prozentzeichen für eine fortlaufende Nummerierung der Initialisierer einer Klasse steht.

Abhängigkeit(en) auf Klassenebene

Führt die Abhängigkeit zu einer Klasse, so wird die Abhängigkeit nur auf Klassen- und Summary-Ebene gespeichert:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaLoeschenK	T_CLASS	Firma	T_CLASS
Class	D_CAST	FirmaLoeschenK	T_CLASS	IFirma	T_CLASS
Member	-	-	-	-	-

Bei der auf Klassenebene eingefügten Abhängigkeit wird eine Referenz zum Knoten des Instanz-Initialisierers gespeichert.

Abhängigkeit(en) zu allen implementierten Konstruktoren:

Der Code eines Instanz-Initialisierers wird implizit beim Erzeugen einer Instanz vor dem Code jedes anderen Konstruktors der Klasse ausgeführt. Man kann sich das als impliziten Aufruf des Initialisierers zu Beginn des Konstruktor-Codes vorstellen. Hierfür ist eine Abhängigkeit D_INIT vorgesehen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	-	-	-	-	-
Class	-	-	-	-	-
Member	D_INIT	FirmaLoeschenK()	T_CONSTR	instance_init_1()	T_INIT

Da es sich um eine klasseninterne Abhängigkeit handelt, wird die Abhängigkeit nur auf Member-Ebene gespeichert.

B.30 Abhängigkeiten zu vererbten Methoden und Attributen

B.30.1 Erläuterung und Beispiel:

Beim Zugriff auf ein vererbtes Attribut bzw. eine vererbte Methode wird von Together eine Referenz auf die Klasse geliefert, in der die Komponente definiert wurde:

```
public class HinweisNachricht extends SeminarisMeldung {
   // static field ERROR is defined in superclass SeminarisMeldung
}
```

```
public class FirmaLoeschenK {
   ...
   public SeminarisMeldung pruefeLoeschen(...) {
      SeminarisMeldung meldung;
      meldung = HinweisNachricht.erzeuge("",
                "Die Firma kann leider nicht gelöscht werden,\n" +
                "da sie noch Seminarveranstaltung(en) gebucht hat.",
                HinweisNachricht.ERROR);
   }
   ...
}
```

B.30.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	FirmaLoeschenK	T_CLASS	SeminarisMeldung	T_CLASS
Class	D_USES_ST	FirmaLoeschenK	T_CLASS	SeminarisMeldung	T_CLASS
Member	D_USES_ST	pruefeLoeschen()	T_METHOD	ERROR	T_FIELD

B.31 Manuelle Together-Abhängigkeiten

B.31.1 Erläuterung und Beispiel:

Together bietet die Möglichkeit “manuelle" Abhängigkeiten zwischen zwei Diagrammelementen (Pakete, Klassen) durch Einfügen einer gerichteten Kante zu modellieren. Diese Abhängigkeiten werden durch einen entsprechenden Kommentar im Sourcecode repräsentiert. Verläuft diese manuelle Abhängigkeit zwischen zwei Klassen, wird eine Abhängigkeit D_MANU definiert:

```
public class Firma {
   ...
   /** @link dependency */
   /*#ErweitertPersistent lnkErweitertPersistent;*/
}
```

B.31.2 Modellierung der Abhängigkeit im Abhängigkeitsgraphen:

Ebene	Abhängigkeitstyp	Von	Knotentyp	Nach	Knotentyp
Summary	D_DEP	Firma	T_CLASS	ErweitertPersistent	T_CLASS
Class	D_MANU	Firma	T_CLASS	ErweitertPersistent	T_CLASS
Member	-	-	-	-	-

Einschränkungen

Initialisierung von Attributen durch anonyme Klassen

In der Version 6.0 erkennt Together nicht, wenn anonyme Klassen zur Initialisierung eines Attributes einer Klasse verwendet werden. Alle Abhängigkeiten, die in dieser anonymen Klasse definiert werden, bleiben derzeit unberücksichtigt. Ein "Workaround" könnte an dieser Stelle nur mit erheblichem Aufwand implementiert werden. Die Initialisierung einer lokalen Variable mit einer anonymen Klasse ist hiervon nicht betroffen.

Initialisierung von Arrays

Ab Version 6.0 liefert Together bei der Initialisierung eines Arrays mittels `new`-Operator innerhalb einer Methode beim Durchlaufen des Methodenrumpfes auch eine Referenz auf diese Initialisierungs-Elemente. Die Referenzen innerhalb der Initialisierung werden derzeit doppelt erkannt. Die mehrfache Speicherung der Abhängigkeiten erfolgt allerdings nur auf Member-Ebene.

Zeilenumbruch in Import-Anweisungen

Befindet sich in einer `import`-Anweisung ein Zeilenumbruch, so ist Together nicht in der Lage, Abhängigkeiten zu der mit diesem Statement importierten Klasse und deren Elementen zu erkennen. Befindet sich also beispielsweise im Rumpf der Klasse der Aufruf einer Methode einer dergestalt importierten Klasse, wird vom Together Open API als Referenz auf diese Methode ein `null`-Wert geliefert. Gleichwohl verläuft der Kompiliervorgang auch bei Zeilenumbrüchen ohne Probleme.

Umlaute in Bezeichnern

Die Verwendung unterschiedlicher Parser beim Kompiliervorgang und beim Durchlaufen des Sourcecodes im Together Open API zeigt sich auch bei der Verwendung von Umlauten in Bezeichnern. Laut Java-Spezifikation sind Unicode-Bezeichner zulässig und die Kompilierung erfolgt in diesen Fällen auch ohne Fehler in Together. Beim Parsen des Projektes im Rahmen der Suche nach Abhängigkeiten liefert das Together Open API jedoch Fehlermeldungen über syntaktisch fehlerhaften Code.

Zusammenfassung

Abschließend soll ein Überblick über alle in den vorausgehenden Abschnitten definierten Abhängigkeiten gegeben werden. Die nachfolgende Graphik ordnet die Abhängigkeiten ferner den Ebenen des Graphen zu, in denen sie auftreten können. Die selbe Zuordnung erfolgt auch für die möglichen Komponenten des Abhängigkeitsgraphen:

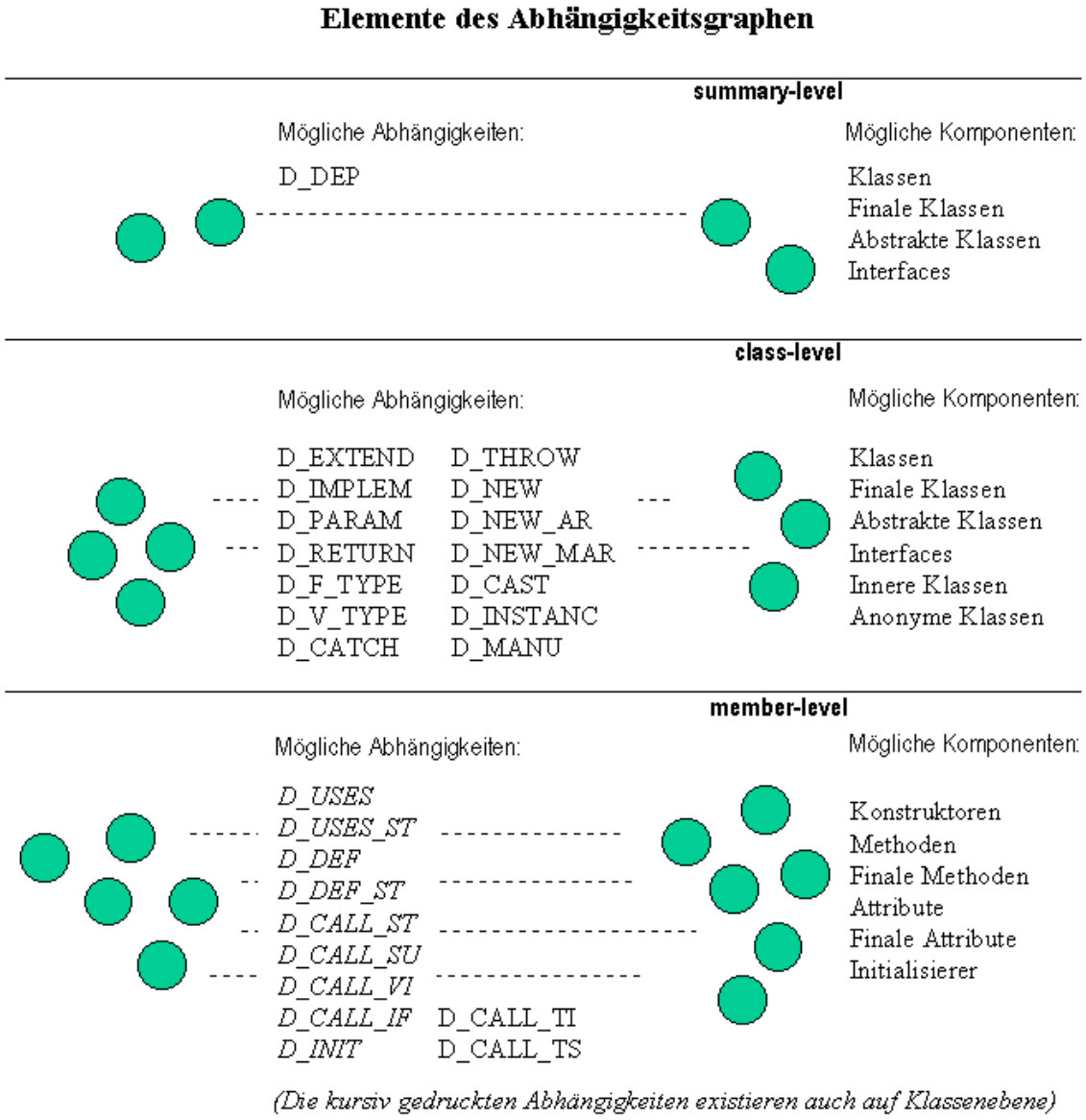


Abbildung 2: Zusammenfassender Überblick über Abhängigkeiten und Komponenten im Graph

Design2Test

Benutzerhandbuch

Version 1.0 (16.03.2003)

Inhaltsverzeichnis

1 INSTALLATION 165

2 SCHNELLSTART (QUICK-TOUR) 166

3 METRIKAUSWAHL 167

4 ANZEIGE DER METRIKEN 169

4.1 ABHÄNGIGKEITSMETRIKEN 169
4.2 KOMPONENTENMETRIKEN 170
4.3 FUNKTIONEN IN DEN METRIKANZEIGEN 171
4.3.1 Navigation zum Sourcecode 171
4.3.2 Sortieren der Ergebnisse 171
4.3.3 Verknüpfungen von Abhängigkeits- und Komponentenmetriken 171
4.3.4 Anzeige im Klassen-Diagramm 171
4.3.5 Neuberechnung der Metriken 172
4.3.6 Beschreibung von Metriken 172
4.3.7 Graphische Anzeige 172
4.3.8 Anzeige der Projektmetriken 172
4.3.9 Ausgabeformat nummerischer Metriken 173
4.3.10 Export der Metrikergebnisse 173
4.3.11 Druck der Metrikergebnisse 173
4.4 PROJEKTMETRIKEN 174
4.4.1 Projektmetriken für das Gesamtprojekt 174
4.4.2 Projektmetriken nach Beseitigung von Abhängigkeiten 174

5 GRAPHISCHE ANZEIGE DER ABHÄNGIGKEITEN 175

5.1 DARSTELLUNG 175
5.2 FUNKTIONEN IN DER GRAPHANZEIGE 176
5.2.1 Aktualisierung der Anzeige 176
5.2.2 Verändern der Farbeinstellungen 176
5.2.3 Navigation zum Sourcecode 177
5.2.4 Anzeige im Klassen-Diagramm 177
5.2.5 Anzeige des qualifizierten Namens 178
5.2.6 Export des Graphen 178
5.2.7 Druck der Graphanzeige 178

6 ERWEITERTE FUNKTIONEN UND EINSTELLUNGEN 179

6.1 METRIKAUSWAHL 179
6.1.1 Setzen der Default-Metrikauswahl 179
6.1.2 Speichern einer Metrikauswahl 179
6.1.3 Einlesen einer gespeicherten Metrikauswahl 180
6.1.4 Optionen für die Metrikberechnung 180
6.2 METRIKANZEIGE 183
6.2.1 Speichern der Metrikergebnisse 183
6.2.2 Druck der Metrikergebnisse 184
6.3 GRAPHANZEIGE 185
6.3.1 Speichern des Abhängigkeitsgraphen im XML-Format 185
6.3.2 Speichern des Abhängigkeitsgraphen im Grafikformat 185
6.3.3 Druck der Graphanzeige 186

ANHANG KONTEXTMENÜS - KURZÜBERSICHTEN 187

1 Installation

Grundlegende Voraussetzung für die Installation von Design2Test ist eine bereits installierte Version des Together Control Centers in der Version 6.0 oder höher. Das Installationsverzeichnis, das Sie bei der Installation von Together festgelegt haben (z.B. `C:\Programme\Together`) wird im Folgenden als `$TGH$` bezeichnet.

Together erwartet alle Module – egal ob Eigen- oder Fremdentwicklung – in einem fest vorgegebenen Pfad. Dieser Pfad lautet `$TGH$\modules\com\togethersoft\modules` und wird nachstehend als `$TGH_MODULES$` abgekürzt.

Design2Test steht Ihnen in zwei Varianten zur Verfügung: Als selbstextrahierende komprimierte Datei (`d2t.zip`) für alle Windows-Systeme und als komprimiertes Archiv (`d2t.tar.Z`) für UNIX/Linux-Plattformen.

Die Installation der verschiedenen Versionen unterscheidet sich bei beiden Plattformen nicht wesentlich:

Windows:

1. Kopieren Sie die Datei `d2t.zip` in den Ablageort der Together-Module `$TGH_MODULES$`.
2. Starten Sie den Extraktionsvorgang durch Doppelklick oder rechten Mausklick auf die gezippte Datei `d2t.zip`.
3. Entpacken Sie das Archiv in das aktuelle Verzeichnis `$TGH_MODULES$`.
4. Löschen Sie die Kopie der Datei `d2t.zip` aus dem Verzeichnis `$TGH_MODULES$`.

UNIX/Linux:

1. Kopieren Sie die Datei `d2t.tar.Z` in den Ablageort der Together-Module `$TGH_MODULES$`.
2. Wechseln Sie in dieses Verzeichnis (achten Sie darauf, dass Sie Schreibrechte für dieses Verzeichnis besitzen müssen).
3. Entkomprimieren Sie das Archiv durch Eingabe von `uncompress d2t.tar.Z`. Sie erhalten eine unkomprimierte Datei `d2t.tar`.
4. Entpacken Sie das Archiv durch Eingabe von `tar xf d2.tar`.
5. Löschen Sie die Archivdatei `d2.tar` im Verzeichnis `$TGH_MODULE$`.

Auf beiden Systemen sollte nach den oben beschriebenen Schritten im Verzeichnis `$TGH_MODULES$` ein Verzeichnis `design2test` existieren. Damit ist die Installation bereits abgeschlossen und das Modul wird beim nächsten Start von Together geladen. Sie erkennen dies daran, dass Ihnen nach dem Start von Together der Menüpunkt **Tools | Testability Metrics ...** zur Verfügung steht.

ACHTUNG: Um die Druckmenüs von Design2Test nutzen zu können, muss das Quality Assurance (QA)-Modul von Together geladen sein. Sie erkennen dies an den Menüpunkten **Tools | Audits ...** und **Tools | Metrics ...** im Hauptmenü von Together. Sind diese Menüpunkte nicht vorhanden, schlagen Sie bitte im *User Guide* des Together Control Centers nach, wie Sie das QA-Modul nachladen können.

2 Schnellstart (Quick-Tour)

Zur Berechnung von Testbarkeitsmetriken wird zunächst der Sourcecode von ausgewählten Komponenten eines Projekts auf das Vorhandensein von Abhängigkeiten zu Komponenten innerhalb und ggf. auch außerhalb des Projektes analysiert. Für die bei der Analyse des Sourcecodes gefundenen Abhängigkeiten erfolgt anschließend die Berechnung und Anzeige der ausgewählten Testbarkeitsmetriken.

Um eine Berechnung durchzuführen, gehen Sie wie folgt vor:

1. Öffnen Sie das Projekt, das Sie analysieren möchten.
2. Wählen Sie im Hauptmenü den Menüpunkt **Tools | Testability Metrics ...**
3. Aktivieren Sie in dem angezeigten Dialog die Metriken, die Sie berechnen lassen möchten. Zu jeder Metrik wird Ihnen im unteren Teil des Dialogs eine kurze Erläuterung angezeigt. Sie können auch eine Standard-Vorbelegung oder eine von Ihnen früher gespeicherte Kombination wählen (→ *3 Metrikauswahl*).
4. Möchten Sie Pakete oder Klassen von der Analyse ausschließen, so können Sie dies in einem weiteren Dialog vornehmen, den Sie durch Betätigen der Schaltfläche **Options** erreichen. Wollen Sie Pakete und Klassen außerhalb des Projekts als Ziel von Abhängigkeiten zulassen, so können Sie dies ebenfalls in diesem Dialog tun (→ *6.1.4 Optionen für die Metrikberechnung*).
5. Nachdem Sie Ihre Auswahl getroffen haben, betätigen Sie die Schaltfläche **Start**.
6. Haben Sie Metriken aus der Kategorie Abhängigkeitsmetriken ausgewählt, so werden Ihnen die Ergebnisse der Berechnung in einem Unterpanel **Dependency Metrics** in der Message Pane von Together angezeigt. Im rechten Teil des Anzeige-Panels werden Ihnen zu einer Abhängigkeit die Einzelabhängigkeiten angezeigt. Über ein Kontextmenü stehen Ihnen weitere Aktionen zur Verfügung (→ *4.1 Abhängigkeitsmetriken*).
7. Haben Sie Metriken aus der Kategorie Komponentenmetriken ausgewählt, so werden Ihnen die Ergebnisse der Berechnung in einem Unterpanel **Component Metrics** in der Message Pane von Together angezeigt. Über ein Kontextmenü stehen Sie Ihnen auch hier weitere Aktionen zur Verfügung (→ *4.2 Komponentenmetriken*).

Achten Sie bitte bei der Berechnung der Metriken darauf, dass alle Ergebnisse nur dann zuverlässig sind, wenn Ihr Sourcecode kompilierfähig ist. Enthält Ihr Programm Syntax-Fehler oder sind nicht alle Bibliotheken oder Pfade eingebunden, so kann dies zu Problemen und fehlerhaften Metrikwerten führen.

3 Metrikauswahl

Zu Beginn einer Metrikberechnung legen Sie die zu berechnenden Metriken und die Rahmenbedingungen der Sourcecode-Analyse fest:

1. Wählen Sie im Hauptmenü den Menüpunkt **Tools | Testability Metrics...**. Sie erhalten folgenden Dialog angezeigt (in Abhängigkeit von den verfügbaren Metriken):

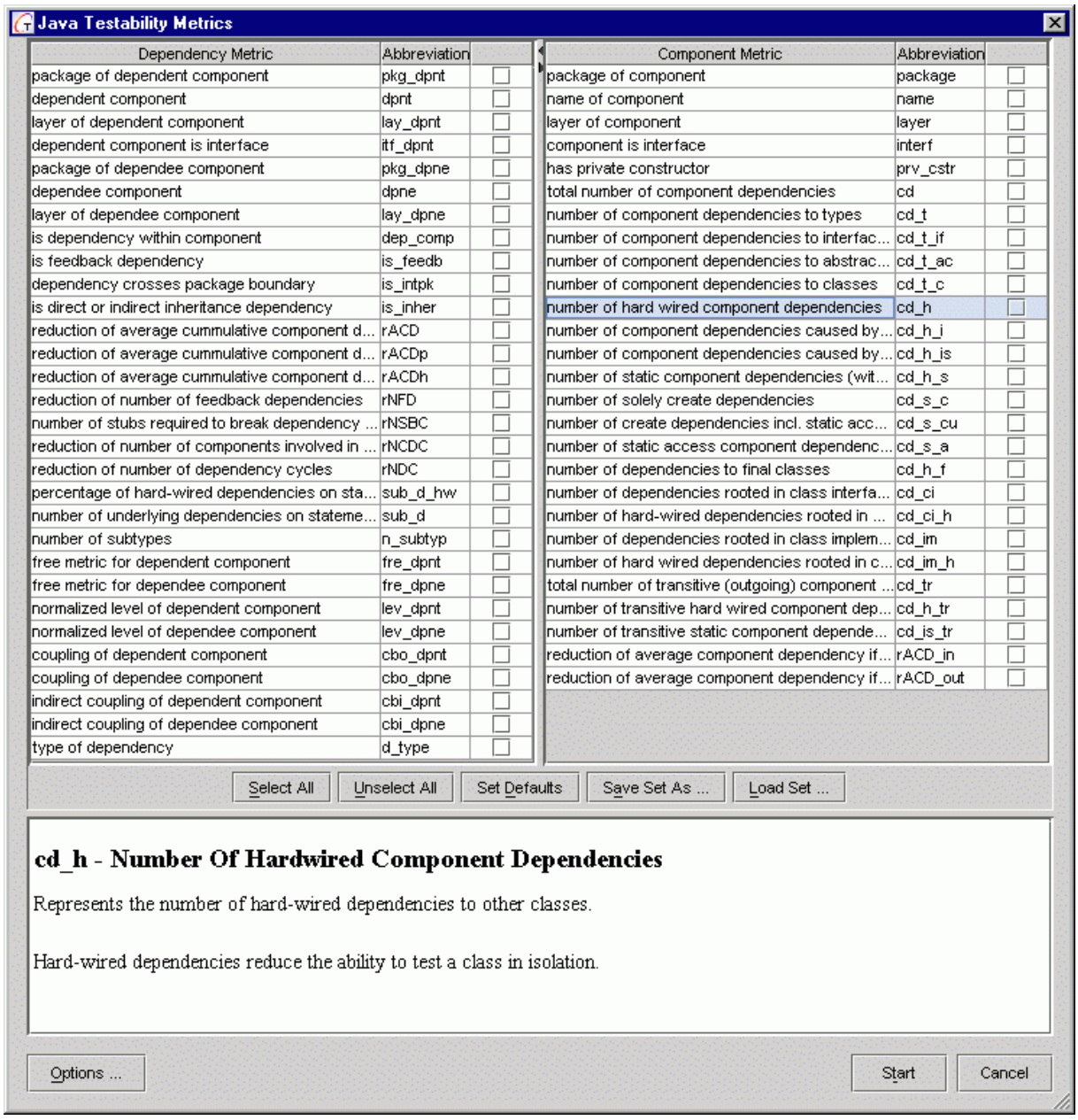


Abbildung 1: Testability Metrics Dialog

Im linken oberen Teil des Dialogs erhalten Sie eine Auflistung der verfügbaren Abhängigkeitsmetriken, im rechten oberen Teil eine Auflistung der verfügbaren Komponentenmetriken.

2. Wählen Sie die Metriken, die Sie berechnet und angezeigt haben möchten, indem Sie in der jeweils letzten Spalte der Metriktabellen das Auswahlfeld aktivieren. Um zu einer der Metriken im unteren Teil des Dialogs eine kurze Erläuterung zu erhalten, klicken Sie bitte einfach in die zugehörige Zeile der Tabelle.

   Für die Berechnung von Abhängigkeitsmetriken wie auch für die Berechnung von Komponentenmetriken muss in der jeweiligen Tabelle mindestens eine der Metriken ausgewählt sein. Haben Sie keine Auswahl getroffen, erhalten Sie einen entsprechenden Hinweis.

   Die beiden Schaltflächen **Select All** und **Unselect All** können Sie zur Auswahl aller Metriken bzw. dem kompletten Zurücksetzen einer durchgeführten Auswahl nutzen.

   Eine Beschreibung der während der Metrikauswahl zusätzlich vorhandenen Funktionalität, wie das Setzen, Speichern und Laden von Metriken sowie der Analyseoptionen, erhalten Sie in den Abschnitten *6.1.1 Setzen der Default-Metrikauswahl*, *6.1.2 Speichern einer Metrikauswahl* und *6.1.3 Einlesen einer gespeicherten Metrikauswahl* des Kapitels *6 Erweiterte Funktionen und Einstellungen*.

3. Sobald Sie mit der Auswahl der Metriken fertig sind und auch keine weiteren Einstellungen vornehmen wollen, klicken Sie auf die Schaltfläche **Start**. Damit wird die Berechnung der Metriken angestoßen.

4 Anzeige der Metriken

4.1 Abhängigkeitsmetriken

Haben Sie Metriken aus der Kategorie Abhängigkeitsmetriken ausgewählt, so erhalten Sie nach dem Ende der Berechnung in der *Message Pane* von Together ein Unterpanel **Dependency Metrics** angezeigt:

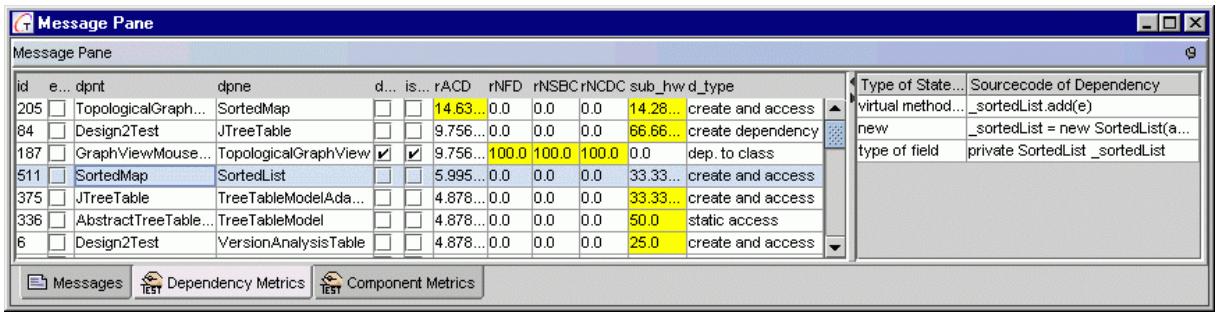

Message Pane
Message Pane
id e... dpnt dpne d... is... rACD rNFD rNSBC rNCDC sub_hw d_type
205 TopologicalGraph... SortedMap 14.63... 0.0 0.0 0.0 14.28... create and access
84 Design2Test JTreeTable 9.756... 0.0 0.0 0.0 66.66... create dependency
187 GraphViewMouse... TopologicalGraphView 9.756... 100.0 100.0 100.0 0.0 dep. to class
511 SortedMap SortedList 5.995... 0.0 0.0 0.0 33.33... create and access
375 JTreeTable TreeTableModelAda... 4.878... 0.0 0.0 0.0 33.33... create and access
336 AbstractTreeTable... TreeTableModel 4.878... 0.0 0.0 0.0 50.0 static access
6 Design2Test VersionAnalysisTable 4.878... 0.0 0.0 0.0 25.0 create and access
Type of State... Sourcecode of Dependency
virtual method... _sortedList.add(e)
new _sortedList = new SortedList(a...
type of field private SortedList _sortedList
Messages
Dependency Metrics
Component Metrics

Abbildung 2: Panel Dependency Metrics

Im linken Teil der geteilten Anzeige erhalten Sie die errechneten Metriken in Tabellenform angezeigt. In den Zeilen der Tabelle sehen Sie die ermittelten und bewerteten Abhängigkeiten. In den Spalten werden Ihnen zu jeder Abhängigkeit die errechneten Werte der Metriken angezeigt, die von Ihnen im Startdialog ausgewählt wurden. Auffällige Metrikwerte sind in der Tabelle farbig hinterlegt.

Im rechten Teil der Anzeige werden Ihnen zu der auf der linken Seite ausgewählten Abhängigkeit die Stellen des Sourcecodes angezeigt, die Auslöser für die Abhängigkeit zwischen den Komponenten sind. Die Anzeige erfolgt ebenfalls in Tabellenform. Zu jeder auslösenden Abhängigkeit wird der Typ der Abhängigkeit und der zugehörige Ausschnitt aus dem Sourcecode angezeigt. Sobald Sie im linken Teil der Anzeige eine andere Abhängigkeit selektieren, wird die Anzeige im rechten Teil automatisch aktualisiert.

Zwischen den beiden Unterfenstern (Metriktabelle und Tabelle der Einzelabhängigkeiten) können Sie zwei kleine Pfeile erkennen. Mit Hilfe dieser beiden Pfeile können Sie wahlweise eines der beiden Fenster ausblenden und das jeweils andere Fenster auf die komplette Breite des Panels **Dependency Metrics** vergrößern.

Um ein Kontextmenü angezeigt zu bekommen, das Ihnen die Funktionalität anbietet, die in der jeweiligen Anzeige zur Verfügung steht, klicken Sie mit der rechten Maustaste in eine der beiden Tabellen. Eine detaillierte Beschreibung der Ihnen zur Verfügung stehenden Funktionen finden Sie in Abschnitt *4.3 Funktionen*. Eine Übersicht und Kurzbeschreibung des Kontextmenüs erhalten Sie in *Anhang Kontextmenüs - Kurzübersichten*.

4.2 Komponentenmetriken

Haben Sie Metriken aus der Kategorie Komponentenmetriken ausgewählt, so werden Ihnen die Ergebnisse der Berechnung in einem Unterpanel **Component Metrics** der Message Pane von Together angezeigt:

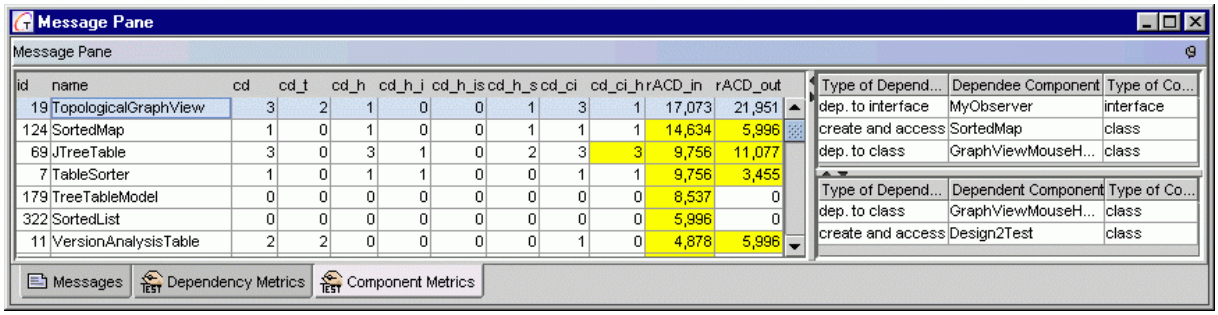

Message Pane
Message Pane
id name cd cd_t cd_h cd_h_i cd_h_is cd_h_s cd_ci cd_ci_h rACD_in rACD_out
19 TopologicalGraphView 3 2 1 0 0 1 3 1 17,073 21,951
124 SortedMap 1 0 1 0 0 1 1 1 14,634 5,996
69 JTreeTable 3 0 3 1 0 2 3 3 9,756 11,077
7 TableSorter 1 0 1 1 0 0 1 1 9,756 3,455
179 TreeTableModel 0 0 0 0 0 0 0 0 8,537 0
322 SortedList 0 0 0 0 0 0 0 0 5,996 0
11 VersionAnalysisTable 2 2 0 0 0 0 1 0 4,878 5,996
Type of Depend... Dependee Component Type of Co...
dep. to interface MyObserver interface
create and access SortedMap class
dep. to class GraphViewMouseH... class
Type of Depend... Dependent Component Type of Co...
dep. to class GraphViewMouseH... class
create and access Design2Test class
Messages
Dependency Metrics
Component Metrics

Abbildung 3: Component Metrics Panel

Im linken Teil der geteilten Anzeige erhalten Sie die errechneten Metriken in Tabellenform angezeigt. In den Zeilen der Tabelle erhalten Sie sämtliche Komponenten des analysierten Projektes aufgelistet. In den Spalten der Tabelle werden Ihnen die zugehörigen Werte der Metriken angezeigt, die von Ihnen im Startdialog ausgewählt wurden. Auffällige Metrikwerte sind in der Tabelle farbig hinterlegt.

Im rechten Teil der Anzeige werden Ihnen zu jeder auf der linken Seite ausgewählten Komponente alle Komponenten angezeigt, die mit dieser Komponente direkt in Abhängigkeit stehen. Dabei sehen Sie in der oberen Hälfte die Komponenten, von denen die aktuell selektierte Komponente abhängig ist. Im unteren Teil werden die Komponenten angezeigt, die von der aktuell selektierten Komponente abhängig sind. Die Anzeige erfolgt jeweils in Tabellenform. Zu jeder verbundenen Komponente wird der Typ der Abhängigkeit, der Name der Komponente und der Typ der Komponente angezeigt. Sobald Sie im linken Teil der Anzeige eine andere Komponente selektieren, wird die Anzeige der verbundenen Komponenten im rechten Teil automatisch aktualisiert.

Zwischen den jeweiligen Teilfenstern (Metriktabelle und Komponententabellen) können Sie zwei kleine Pfeile erkennen. Mit Hilfe dieser beiden Pfeile können Sie wahlweise einen der Fensterbereiche ausblenden und den jeweils anderen Bereich vergrößern.

Um ein Kontextmenü angezeigt zu bekommen, das Ihnen die Funktionalität anbietet, die in der jeweiligen Anzeige zur Verfügung steht, klicken Sie mit der rechten Maustaste in eine der beiden Tabellen. Eine detaillierte Beschreibung der Ihnen zur Verfügung stehenden Funktionen finden Sie in Abschnitt *4.3 Funktionen*. Eine Übersicht und Kurzbeschreibung des Kontextmenüs erhalten Sie in *Anhang Kontextmenüs - Kurzübersichten*.

4.3 Funktionen in den Metrikanzeigen

4.3.1 Navigation zum Sourcecode

Die Anzeige der Abhängigkeits- und Komponentenmetriken ist direkt mit dem analysierten Sourcecode verknüpft:

- Doppelklicken Sie in den Metriktabellen oder der Komponentenanzeige auf eine Komponente, um diese im Texteditor-Fenster von Together zu öffnen. Oder wählen Sie im Kontextmenü der jeweiligen Komponentenspalte den Menüpunkt **Edit,** um sich den Sourcecode der Komponente im Texteditor anzeigen zu lassen.
- Doppelklicken Sie auf eine der auslösenden Abhängigkeiten im rechten Teil des Panels **Dependency Metrics**, um im Texteditor an die Stelle des Sourcecodes der abhängigen Klasse zu springen, die Ursache der Abhängigkeit ist. Oder wählen Sie im Kontextmenü der auslösenden Abhängigkeiten den Menüpunkt **Go to Source**.

4.3.2 Sortieren der Ergebnisse

Sie können die Metriktabellen nach den Werten jeder einzelnen Spalte sortieren. Klicken Sie hierzu mit der linken (aufsteigende Sortierreihenfolge) oder rechten (absteigende Sortierreihenfolge) Maustaste auf eine der Spaltenüberschriften. Oder wählen Sie alternativ im Kontextmenü der Metriktabellen das Untermenü **Sort** und einen der beiden Einträge **Ascending** (aufsteigend) oder **Descending** (absteigend).

4.3.3 Verknüpfungen von Abhängigkeits- und Komponentenmetriken

Wollen Sie sich aus dem Panel **Dependency Metrics** die Komponentenmetriken zu einer Komponente anzeigen lassen, so klicken Sie mit der rechten Maustaste auf die Komponente und wählen im erscheinenden Kontextmenü den Menüpunkt **Component Metrics**. Daraufhin wird Ihnen das Unterpanel **Component Metrics** (→ *4.2 Komponentenmetriken*) angezeigt. Die Zeile mit den zu dieser Komponente gehörenden Metriken erscheint selektiert.

Wenn Sie sich im Panel **Component Metrics** in der Anzeige der verbundenen Komponenten befinden, können Sie sich die Metrikwerte der zugrunde liegenden Abhängigkeit anzeigen lassen. Wählen Sie hierzu im Kontextmenü der Komponententabelle den Menüpunkt **Dependency Metrics**. Daraufhin wird Ihnen das Unterpanel **Dependency Metrics** (→ *4.1 Abhängigkeitsmetriken*) angezeigt. Die Zeile mit den Metriken der korrespondierenden Abhängigkeit erscheint selektiert.

Im Panel **Component Metrics** besteht ferner die Möglichkeit, sich zu jeder verbundenen Komponente die Komponentenmetriken anzeigen zu lassen. Wählen Sie hierzu im Kontextmenü der über Abhängigkeit verbundenen Komponente den Menüpunkt **Select Component Metrics**. Daraufhin wird die Anzeige der Metriktabelle aktualisiert und die Metrikwerte der verbundenen Komponente erscheinen selektiert. Gleichzeitig erfolgt auch die Aktualisierung der geöffneten Komponentenfenster im rechten Teil der Anzeige. Auf diese Weise können Sie in beliebiger Richtung entlang eines Pfads im Abhängigkeitsgraphen navigieren.

4.3.4 Anzeige im Klassen-Diagramm

Möchten Sie sich das Klassendiagramm für eine an einer Abhängigkeit beteiligten Komponente anzeigen lassen, so wählen Sie in den Metriktabellen oder der Komponentenanzeige den Menüpunkt **Select on Diagram** im Kontextmenü der jeweiligen Komponentenspalte. Es öffnet sich die *Designer Pane* von Together und die betreffende Komponente erscheint innerhalb des Diagramms selektiert.

4.3.5 Neuberechnung der Metriken

Sie können die erneute Berechnung der Metriken über das Kontextmenü der Metriktabellen anstoßen. Es stehen Ihnen hierfür zwei Varianten zur Verfügung:

- Wählen Sie den Menüpunkt **Refresh**, um für sowohl Abhängigkeits- als auch Komponentenmetriken eine neue Metrikberechnung zu starten. Diese Neuberechnung erfolgt auf Basis der zuletzt ausgewählten Metriken und der zuletzt eingestellten Rahmenbedingungen für die Analyse des Sourcecodes.
- Wählen Sie den Menüpunkt **Restart**, um den Auswahldialog für Metriken komplett neu zu starten. Sie können so die Auswahl der Abhängigkeits- und Komponentenmetriken neu vornehmen und die Analyse-Optionen verändern.

4.3.6 Beschreibung von Metriken

Sie können zu jeder Metrik eine kurze Beschreibung erhalten. Klicken Sie hierzu mit der rechten Maustaste in eine der Metrikspalten und wählen Sie aus dem angezeigten Kontextmenü den Menüpunkt **Show Description**. Es wird ein Dialogfenster geöffnet, dem Sie die gewünschten Informationen entnehmen können:

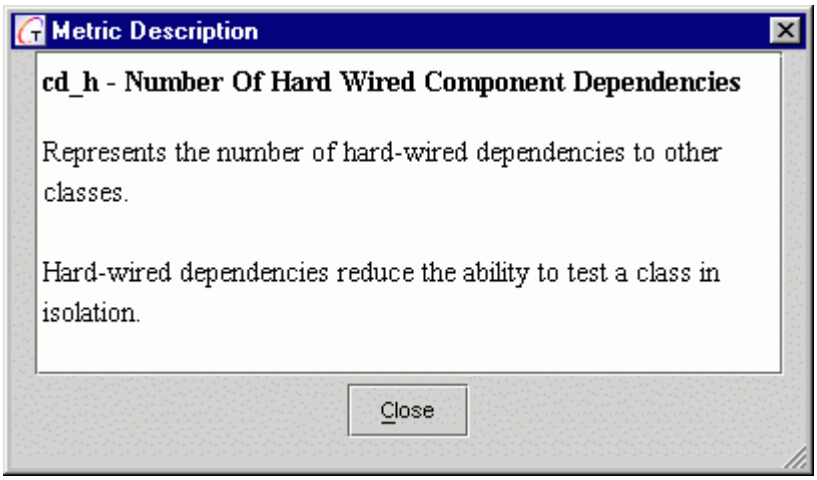

Metric Description
cd_h - Number Of Hard Wired Component Dependencies
Represents the number of hard-wired dependencies to other classes.
Hard-wired dependencies reduce the ability to test a class in isolation.
Close

Abbildung 4: Hilfetext zu einer Metrik

4.3.7 Graphische Anzeige

Wollen Sie eine graphische Darstellung des Abhängigkeitsgraphen unter Berücksichtigung der Metrikergebnisse erhalten, so wählen Sie aus dem Kontextmenü der Metriktabellen den Menüpunkt **Graph**. Eine detaillierte Beschreibung des daraufhin erscheinenden Anzeigefensters finden Sie in Abschnitt *5 Graphische Anzeige der Abhängigkeiten*.

4.3.8 Anzeige der Projektmetriken

Bei den Projektmetriken handelt es sich um Abhängigkeitsmetriken. Die Möglichkeit der Anzeige steht Ihnen deshalb auch nur im Panel **Dependency Metrics** zur Verfügung**.** Wählen Sie hierzu im Kontextmenü der Metriktabelle den Menüpunkt **Project Metrics**. Daraufhin wird ein kleines Fenster geöffnet, in dem Sie die entsprechenden Daten angezeigt bekommen (→ *4.4.1 Projektmetriken für das Gesamtprojekt*).

In der Spalte **exclude** der Metriktabelle steht Ihnen der zusätzliche Menüpunkt **Project Metrics without** zur Verfügung. Wählen Sie die Abhängigkeiten, die Sie entfernt sehen möchten und klicken Sie den Menüpunkt an, um ein kleines Fenster zu erhalten, in dem die Metrikwerte ohne die in der

Spalte **exclude** abgewählten Abhängigkeiten angezeigt werden (→ *4.4.2 Projektmetriken nach Beseitigung von Abhängigkeiten*).

4.3.9 Ausgabeformat nummerischer Metriken

Für einige der Metriken können Sie zwischen der Anzeige der absoluten Metrikwerte und der Anzeige von Prozentwerten wählen. Für Abhängigkeitsmetriken handelt es bei der Prozentangabe um den Grad der Verringerung des korrespondierenden Projekt-Metrikwertes, der durch Entfernung der Abhängigkeit erreicht würde. Für Komponentenmetriken gibt der Prozentwert an, welchen Anteil zu der Gesamtzahl des betreffenden Abhängigkeitstyps ein weiter verfeinerter Typ der Abhängigkeit beiträgt. Um das von Ihnen gewünschte Anzeigeformat einzustellen, wählen Sie im Untermenü **Format** des Kontextmenüs einen der beiden Einträge **Absolute Values** oder **Values as Percentage**.

4.3.10 Export der Metrikergebnisse

Sie können die Ergebnisse der Metriktabellen zur weiteren Verarbeitung im Textformat speichern. Wählen Sie hierzu den Menüpunkt **Export** aus dem Kontextmenü. Eine nähere Beschreibung des Speichervorgangs finden Sie Abschnitt *6.2.1 Speichern der Metrikergebnisse* des Kapitels *6 Erweiterte Funktionen und Einstellungen*.

4.3.11 Druck der Metrikergebnisse

Sie können sich die Ergebnisse der Metriktabellen ausdrucken lassen. Wählen Sie hierzu den Menüpunkt **Print** aus dem Kontextmenü. Eine nähere Beschreibung des Druckdialogs finden Sie in Abschnitt *6.2.2 Druck der Metrikergebnisse* im Kapitel *6 Erweiterte Funktionen und Einstellungen.*

4.4 Projektmetriken

Bei den Projektmetriken handelt es sich um folgende Abhängigkeitsmetriken:

- ACD (**A**verage cummulative **C**omponent **D**ependency)
- NFD (**N**umber of **F**eedback **D**ependencies)
- NSBC (**N**umber of **S**tubs required to **B**reak dependency **C**ycles)
- NCDC (**N**umber of **C**omponents involved in **D**ependency **C**ycles)
- NDC (**N**umber of **D**ependency **C**ycles)

Sie erhalten in den nachfolgend beschriebenen Panels stets nur diejenigen Projektmetriken angezeigt, die Sie während der Metrik-Auswahl (→ *3 Metrikauswahl*) zur Berechnung vorgesehen haben.

4.4.1 Projektmetriken für das Gesamtprojekt

In der Anzeige der Abhängigkeitsmetriken (Panel **Dependency Metrics**) haben Sie die Möglichkeit, sich die Metriken für das Gesamtprojekt anzeigen zu lassen. Wählen Sie hierzu den Menüpunkt **Project Metrics** im Kontextmenü. Es öffnet sich ein kleines Dialogfenster mit den bei der Analyse der Abhängigkeiten errechneten Werten. Neben dem Kürzel der jeweiligen Metrik erhalten Sie in der Spalte **ZeroValue** die Gesamt-Metrikwerte des zuletzt analysierten Projektes:

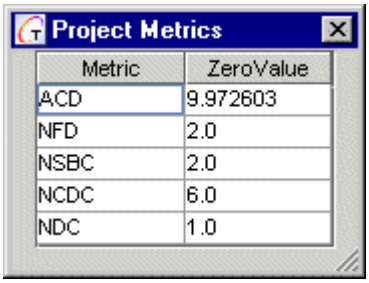


Abbildung 5: Anzeige der Gesamt-Projektmetriken

4.4.2 Projektmetriken nach Beseitigung von Abhängigkeiten

In der Spalte **exclude** der Anzeige der Abhängigkeitsmetriken haben Sie die Möglichkeit, sich die Projektmetriken anzeigen zu lassen, die sich nach Beseitigung der ausgewählten Abhängigkeiten ergeben würden. Wählen Sie hierzu den Menüpunkt **Project Metrics without** des Kontextmenüs. Zur Anzeige öffnet sich ein eigenes Fenster. Zu jeder Metrik erhalten Sie in der Spalte **ZeroValue** die Gesamtmetrikwerte des zuletzt analysierten Projektes, in der Spalte **Value** die Gesamtwerte nach Beseitigung der ausgewählten Abhängigkeiten und in der Spalte **Improvement [%]** die daraus resultierende prozentuale Verringerung des Metrikwertes:

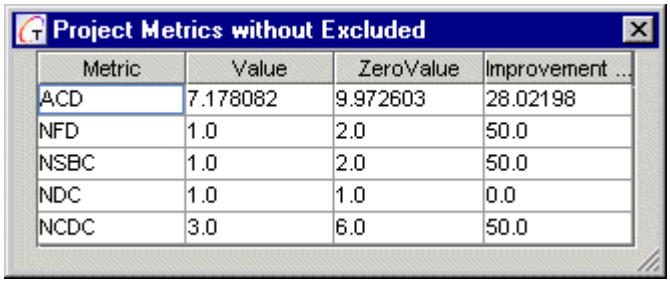


Abbildung 6: Anzeige der Projektmetriken nach Beseitigung ausgewählter Abhängigkeiten

5 Graphische Anzeige der Abhängigkeiten

5.1 Darstellung

Möchten Sie sich den Abhängigkeitsgraphen visualisieren lassen, wählen Sie im Kontextmenü der Metriktabellen (→ *4 Anzeige der Metriken*) den Menüpunkt **Graph**. Es öffnet sich hierauf ein Fenster mit der Darstellung des Abhängigkeitsgraphen als geometrischer Graph in der Ebene:

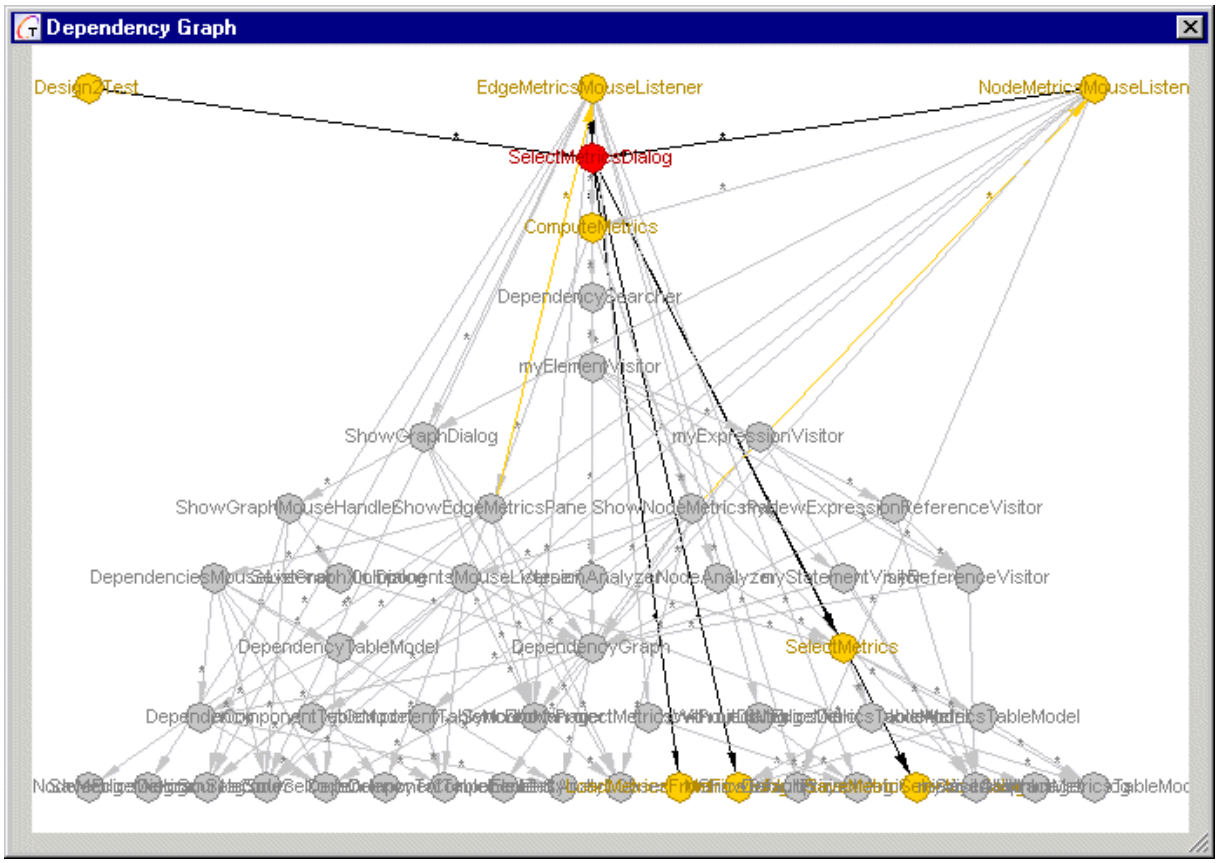


Abbildung 7: Graphische Darstellung des Abhängigkeitsgraphen (Farbschema „Normal“)

Die Knoten des Graphen stellen hierbei die Komponenten des analysierten Projektes dar. Sie sind mit dem Namen der Komponente versehen. Bei den Kanten des Graphen handelt es sich um die zwischen den Komponenten ermittelten Abhängigkeiten. Die Richtung jeder Kante verläuft von der abhängigen Komponente zum Ziel der Abhängigkeit. Sie können die Komponenten über „Drag und Drop“ beliebig verschieben. Klicken Sie hierzu den betreffenden Knoten an und ziehen Sie ihn bei gedrückter Maustaste an die gewünschte Stelle.

Die Anzeige des Graphen ist mit den Metriktabellen synchronisiert. Wählen Sie beispielsweise eine Abhängigkeit im Panel **Dependency Metrics** aus, so wird die entsprechende Kante im Graphen farbig hervorgehoben. Markieren Sie umgekehrt eine Kante in der Graphanzeige, so erscheint die entsprechende Zeile der Metriktabelle selektiert. Analog sind die angezeigten Komponenten mit dem Panel **Component Metrics** verknüpft.

5.2 Funktionen in der Graphanzeige

5.2.1 Aktualisierung der Anzeige

Sie haben die Möglichkeit, die angezeigten Komponenten und Abhängigkeiten automatisch neu ordnen zu lassen, z.B. nachdem Sie das Anzeigefenster in seiner Größe geändert haben. Wählen Sie hierzu aus dem Kontextmenü den Menüpunkt **Layout**. Die vorhandenen Komponenten und Abhängigkeiten werden neu arrangiert und die Anzeige aktualisiert.

5.2.2 Verändern der Farbeinstellungen

Um einen besseren Überblick über verschiedene Aspekte des Abhängigkeitsgraphen und der errechneten Metriken zu gewinnen, können Sie den Graphen mit unterschiedlichen Farbschemata betrachten. Wählen Sie hierzu im Kontextmenü das Untermenü **Set Color Schema** und einen der dort befindlichen Menüpunkte **Normal**, **Show Component**, **Show Dependency Type**.

Haben Sie im Modus **Normal** eine Komponente durch Anklicken ausgewählt, so wird die aktivierte Komponente rot und alle unmittelbar über Abhängigkeit verbundenen Komponenten orange angezeigt. Klicken Sie auf eine durch eine der Kanten repräsentierte Abhängigkeit, so wird diese Abhängigkeit rot und alle über Abhängigkeiten erreichbaren Komponenten des Abhängigkeitsgraphen grau hinterlegt angezeigt. Ferner sind in diesem Modus alle Feedback-Kanten orange gekennzeichnet (→ *Abbildung 7*).

Möchten Sie einen Überblick über alle Zyklen des Abhängigkeitsgraphen gewinnen, so wählen Sie im Untermenü den Menüpunkt **Show Component**. Die existierenden Zyklen werden daraufhin verschiedenfarbig hervorgehoben. Alle an einem Zyklus beteiligten Abhängigkeiten und Komponenten werden in derselben Farbe dargestellt:

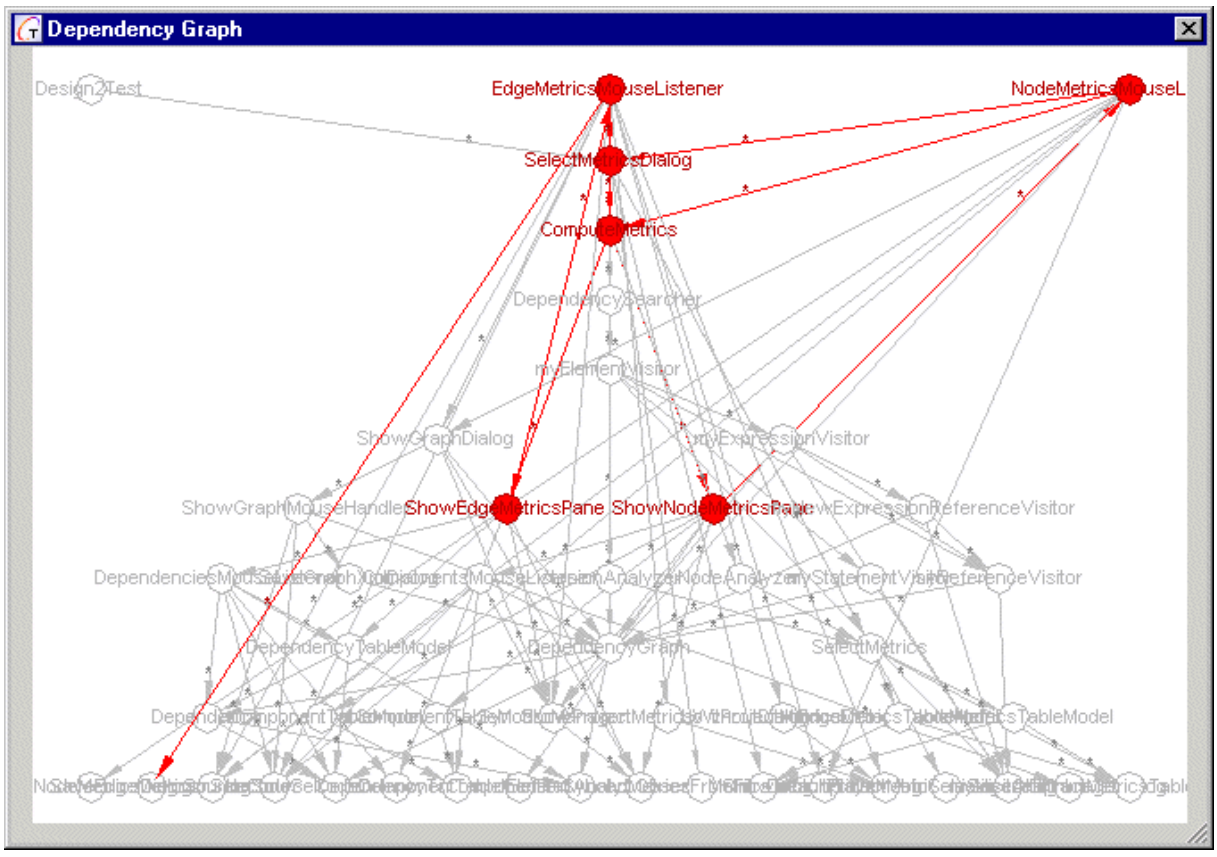


Abbildung 8: Graphanzeige im Farbschema „Show Component“

Um sich einen Überblick über die Typen der Abhängigkeiten zu verschaffen, wählen Sie im Untermenü den Menüpunkt **Show Dependency Type**. Daraufhin werden reinen Vererbungsabhängigkeiten orange, die übrigen fest verdrahteten Abhängigkeiten rot und alle verbleibenden Abhängigkeiten grün dargestellt:

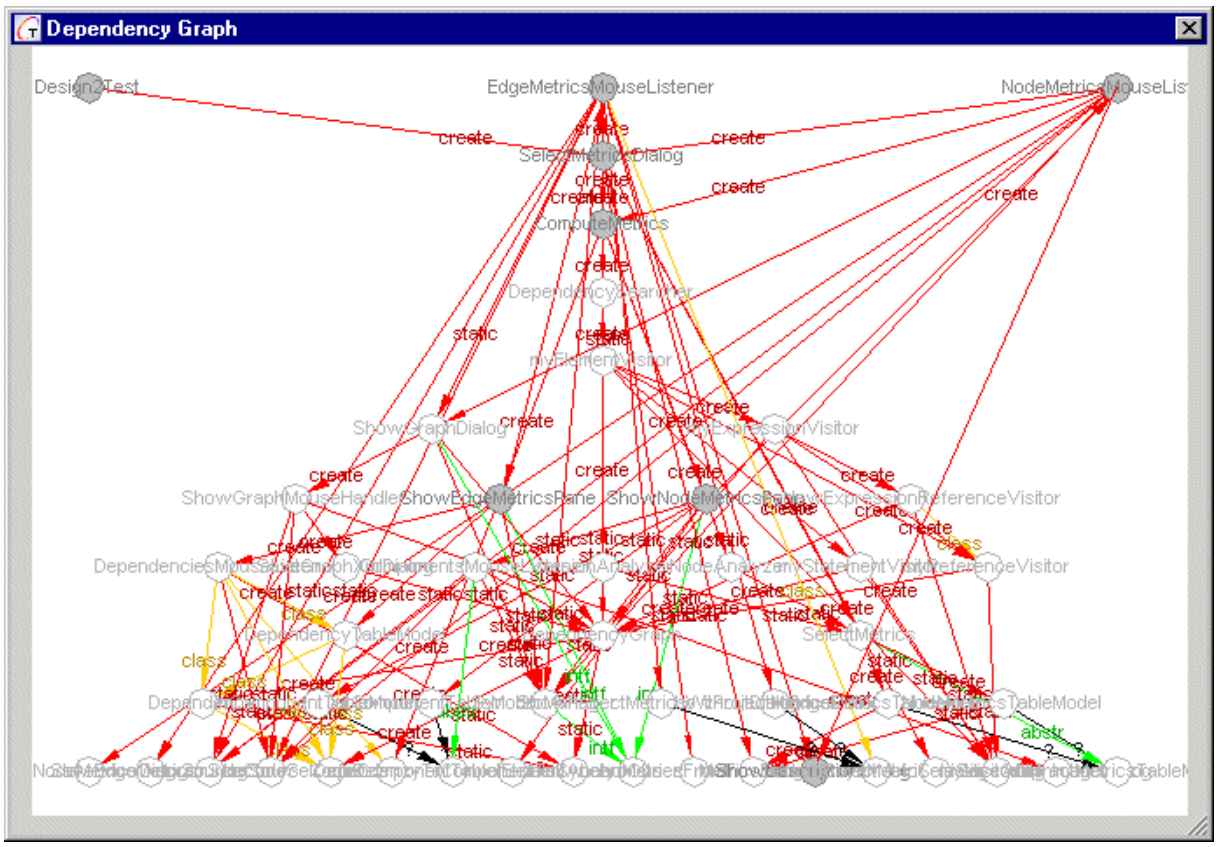


Abbildung 9: Graphanzeige im Farbschema „Show Dependency Type“

5.2.3 Navigation zum Sourcecode

Die Anzeige des Graphen ist mit dem analysierten Sourcecode verknüpft. Klicken Sie mit der rechten Maustaste auf eine der angezeigten Komponenten und wählen Sie im Kontextmenü den Menüpunkt **Edit,** um sich den Sourcecode der Komponente im Texteditor von Together anzeigen zu lassen.

5.2.4 Anzeige im Klassen-Diagramm

Möchten Sie sich das Klassendiagramm für eine an einer Abhängigkeit beteiligten Komponente anzeigen lassen, so klicken Sie mit der rechten Maustaste die betreffende Komponente an und wählen Sie im erscheinenden Kontextmenü den Menüpunkt **Select on Diagram**. Es öffnet sich die *Designer Pane* von Together und die betreffende Komponente erscheint innerhalb des Diagramms selektiert.

5.2.5 Anzeige des qualifizierten Namens

Möchten Sie sich den voll qualifizierten Namen einer der Komponenten anzeigen lassen, so klicken Sie mit der rechten Maustaste die betreffende Komponente an und wählen im Kontextmenü den Menüpunkt **Show Qualified Name...**. Es öffnet sich ein kleines Dialogfenster mit der gewünschten Angabe:

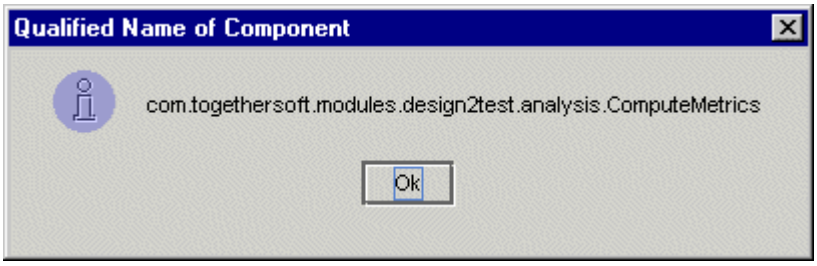


Abbildung 10: Anzeige des voll qualifizierten Namens einer Komponente

5.2.6 Export des Graphen

Sie können den Graph im XML-Format oder in einem Grafikformat (JPEG) zur weiteren Bearbeitung speichern. Wählen Sie hierzu im Kontextmenü das Untermenü **Export** und einen der beiden Menüpunkte **As Image...** oder **As XML...**. Eine nähere Beschreibung des Speichervorgangs finden Sie in Kapitel *6 Erweiterte Funktionen und Einstellungen* in den Abschnitten *6.3.1 Speichern des Abhängigkeitsgraphen im XML-Format* und *6.3.2 Speichern des Abhängigkeitsgraphen im Grafikformat.*

5.2.7 Druck der Graphanzeige

Sie können sich den Graph in der aktuellen Anzeige ausdrucken lassen. Wählen Sie hierzu den Menüpunkt **Print...** aus dem Kontextmenü. Eine nähere Beschreibung des Druckdialogs finden Sie in Abschnitt *6.3.3 Druck der Graphanzeige* von Kapitel *6 Erweiterte Funktionen und Einstellungen.*

6 Erweiterte Funktionen und Einstellungen

6.1 Metrikauswahl

6.1.1 Setzen der Default-Metrikauswahl

Eine empfohlene Metrikauswahl ist Bestandteil des ausgelieferten Moduls. Die zugrunde liegende Datei `defaultMetrics.tms` befindet sich im Unterverzeichnis `config` des Installationsverzeichnisses des Moduls (→ *1 Installation*). Um die darin gespeicherte Metrikauswahl zu setzen, betätigen Sie im Dialog zur Metrikauswahl (→ *3 Metrikauswahl*) die Schaltfläche **Set Defaults**.

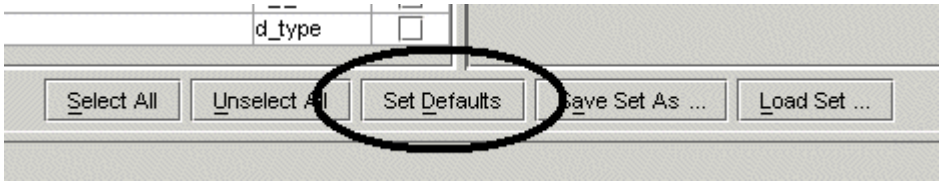


Abbildung 11: Optionen-Leiste im Metrikdialog

6.1.2 Speichern einer Metrikauswahl

Sie haben die Möglichkeit eine von Ihnen vorgenommene Metrikauswahl für eine erneute Verwendung speichern. Klicken Sie hierzu auf die Schaltfläche **Save Set As ...** im Dialog zur Metrikauswahl (→ *3 Metrikauswahl*):

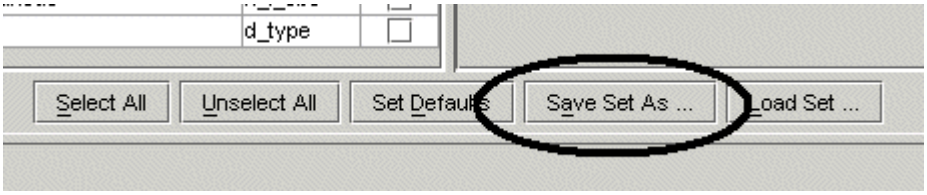


Abbildung 12: Optionen-Leiste im Metrikdialog

Wählen Sie in dem erscheinenden Auswahldialog den gewünschten Speicherort und den gewünschten Namen der Datei:

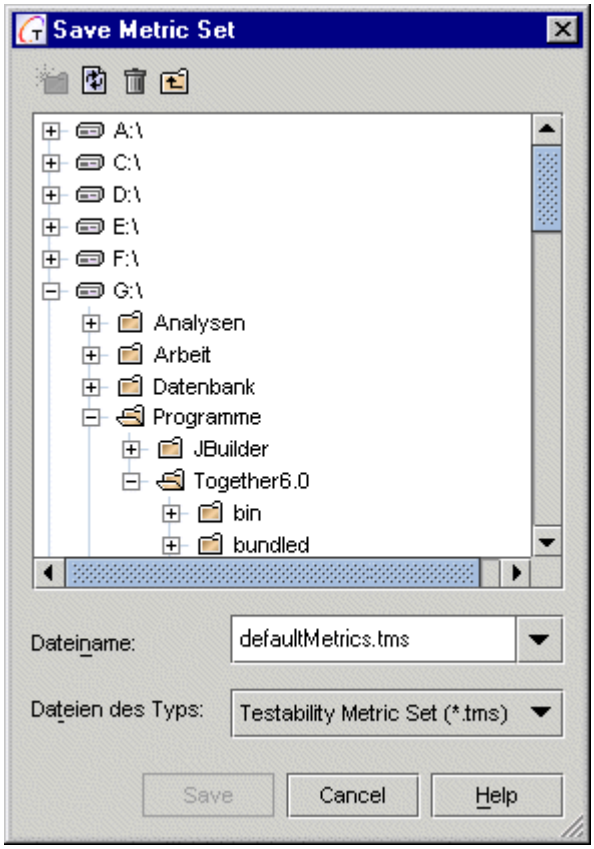


Abbildung 13: Dialog zum Speichern einer Metrikauswahl

Wie in der Abbildung als Beispiel gezeigt, können Sie auf diese Art auch die Standard-Vorbelegung überschreiben, sofern Sie dies möchten.

6.1.3 Einlesen einer gespeicherten Metrikauswahl

Um Ihre gespeicherten Metrikauswahlen wieder abzurufen, betätigen Sie während der Metrikauswahl die Schaltfläche **Load Set ...**:

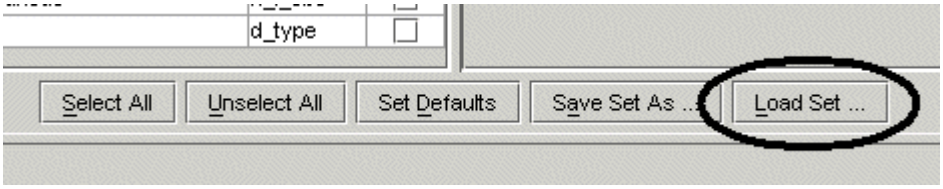


Abbildung 14: Optionen-Leiste im Metrikdialog

Der dort angezeigte Dialog gleicht dem Dialog zum Speichern der Metriken (→ *Abbildung 13*).

6.1.4 Optionen für die Metrikberechnung

Standardmäßig werden

- alle Komponenten des aktiven Projektes nach Abhängigkeiten durchsucht und
- als Ziel von Abhängigkeiten nur die Komponenten des aktiven Projektes berücksichtigt.

Während der Metrikauswahl (→ *3 Metrikauswahl*) besitzen Sie die Möglichkeit, bestimmte Pakete oder Klassen von der Analyse auszuschließen (z.B. Testklassen). Ferner können Sie auf diesem Wege

Pakete und Klassen außerhalb des Projekts (importierte Klassen) als Ziel von Abhängigkeiten zulassen.

Um die Standard-Einstellungen zu verändern, betätigen Sie die Schaltfläche **Options** im linken unteren Teil des Metrik-Dialogs. Es öffnet sich folgender Auswahldialog:

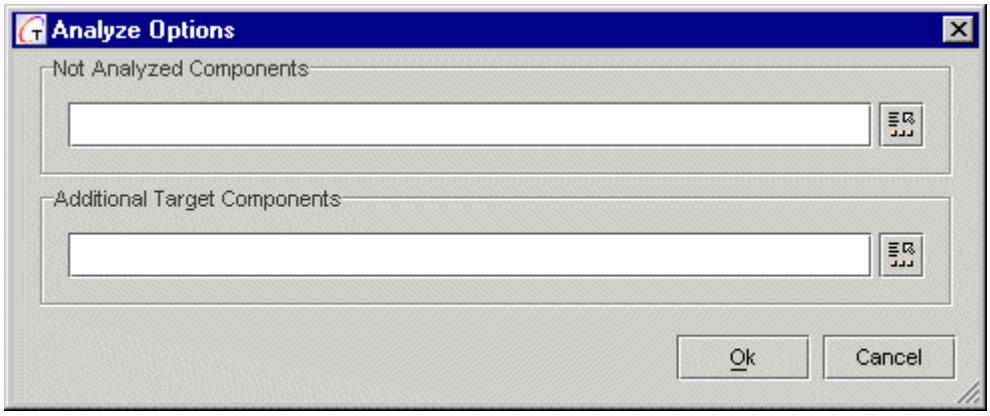


Abbildung 15: Analyze Options Dialog

6.1.4.1 Komponenten von der Analyse ausschließen

Sie haben im oberen Teil des Dialogs (→ *Abbildung 15: Analyze Options Dialog*) die Möglichkeit, bestimmte Klassen und Pakete von der Analyse auszuschließen. Die ausgeschlossenen Komponenten werden weder auf Abhängigkeiten analysiert noch als Ziel von Abhängigkeiten berücksichtigt. Um Komponenten auszuschließen, klicken Sie auf die kleine Schaltfläche neben dem Textfeld. Es öffnet sich folgendes Auswahlfenster:

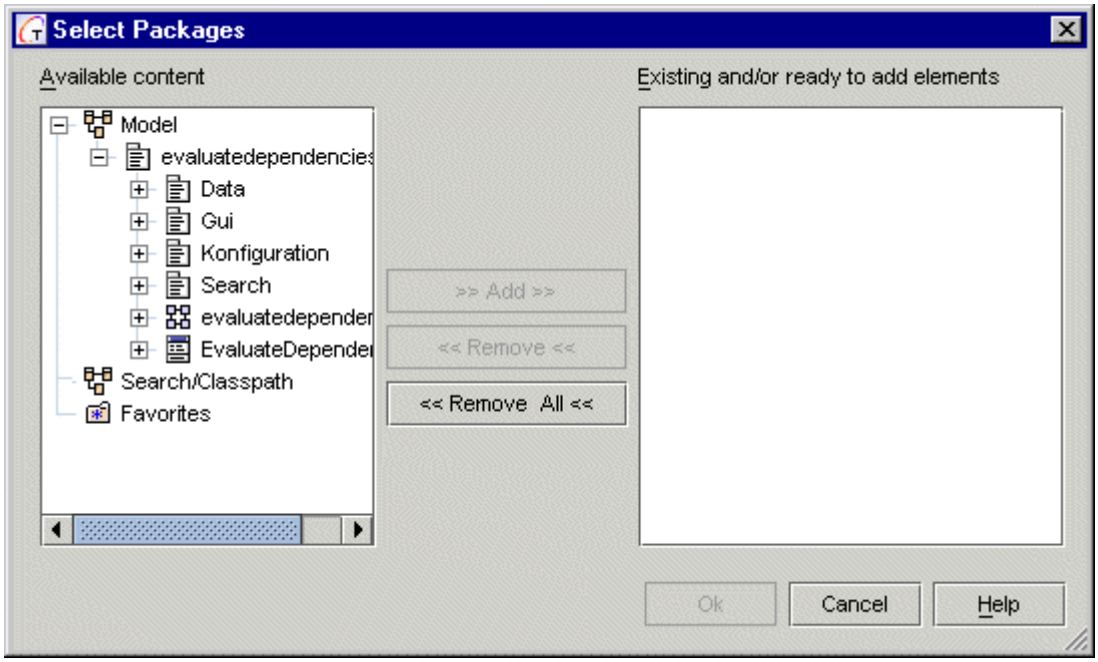


Abbildung 16: Ausschluss von Komponenten

Ihnen werden alle Pakete und Klassen des eigenen Projektes angezeigt. Andere Komponenten (z.B. Diagramme oder Komponenten aus dem Classpath) sind für Sie an dieser Stelle nicht sichtbar. Wählen Sie die Komponenten des Projekts (im Dialog als Model bezeichnet) aus, die nicht nach

Abhängigkeiten durchsucht werden sollen. Betätigen Sie die Schaltfläche **Add**, um die auszuschließenden Komponenten der Liste auf der rechten Seite hinzuzufügen. Sobald Sie Ihre Auswahl getroffen haben, klicken Sie auf die Schaltfläche **Ok**.

6.1.4.2 Komponenten als Ziele von Abhängigkeiten hinzufügen

Im unteren Teil des Dialogs (→ *Abbildung 15: Analyze Options Dialog*) können Sie einzelne Pakete und Klassen, die im Classpath liegen, als Ziel von Abhängigkeiten zusätzlich berücksichtigen lassen. Klicken Sie hierzu auf die kleine Schaltfläche neben dem Textfeld. Es öffnet sich folgendes Auswahlfenster:

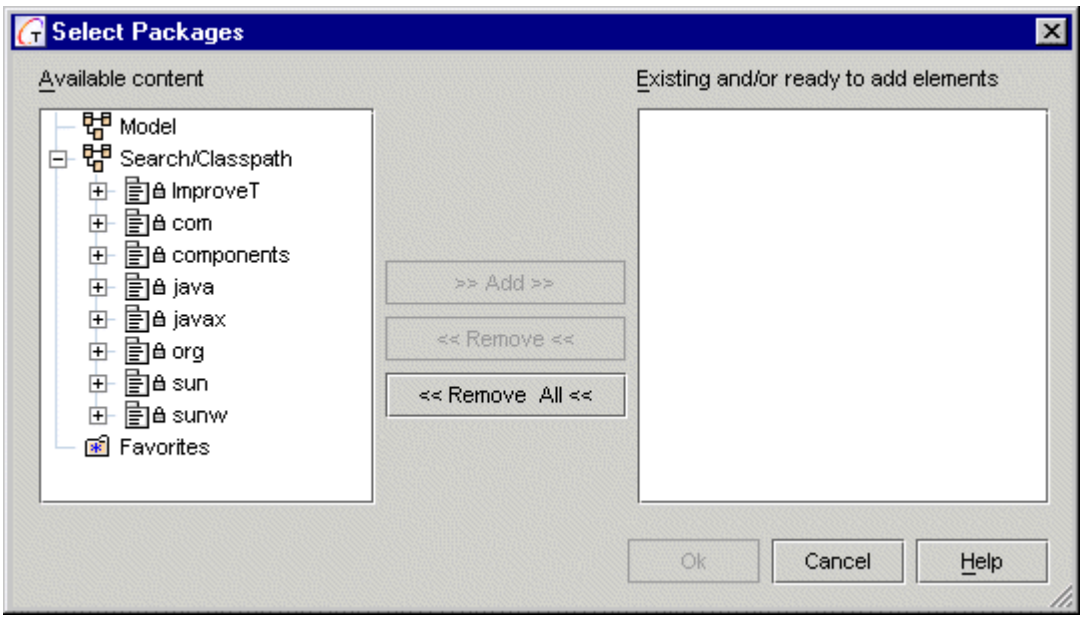


Abbildung 17: Hinzufügen von Zielkomponenten

Ihnen werden alle Pakete und Klassen, die innerhalb des Classpaths liegen, angezeigt. Andere Komponenten (z.B. Diagramme oder Komponenten aus dem Projekt) sind für Sie an dieser Stelle nicht sichtbar. Wählen Sie die Komponenten im Classpath aus, die als Ziel von Abhängigkeiten Berücksichtigung finden sollen. Betätigen Sie die Schaltfläche **Add**, um die zusätzlich zu berücksichtigenden Komponenten der Liste auf der rechten Seite hinzuzufügen. Sobald Sie Ihre Auswahl getroffen haben, klicken Sie auf die Schaltfläche **Ok**.

6.2 Metrikanzeige

6.2.1 Speichern der Metrikergebnisse

Um die Abhängigkeits- oder Komponentenmetriken zur weiteren Verarbeitung im Textformat zu speichern, wählen Sie den Menüpunkt **Export...** im Kontextmenü der Metriktabellen. Hierauf öffnet sich folgender Dialog:

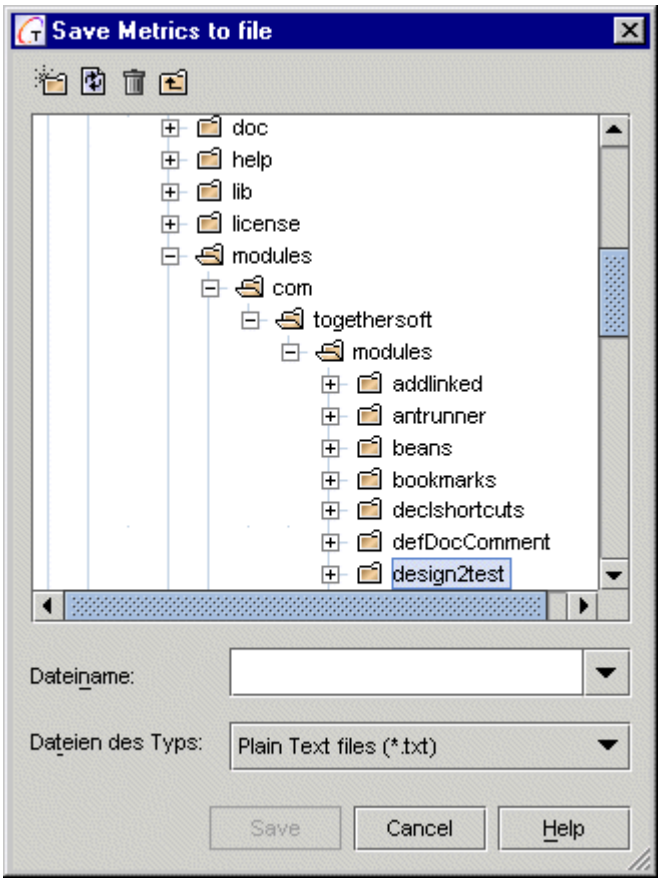


Abbildung 18: Dialog zum Speichern der Metrikergebnisse

Wählen Sie das Verzeichnis und geben Sie den Namen der Datei an, unter dem die Resultate der Metrikberechnung gespeichert werden sollen. Um den Speichervorgang abzuschließen, drücken Sie die Schaltfläche **Save**.

6.2.2 Druck der Metrikergebnisse

ACHTUNG: Um die Druckmenüs von Design2Test nutzen zu können, muss das Quality Assurance (QA)-Modul von Together geladen sein (→ *1 Installation*).

Um die Abhängigkeits- oder Komponentenmetriken auszudrucken, wählen Sie den Menüpunkt **Print...** im Kontextmenü der Metriktabellen. Hierauf öffnet sich folgender Druck-Dialog:

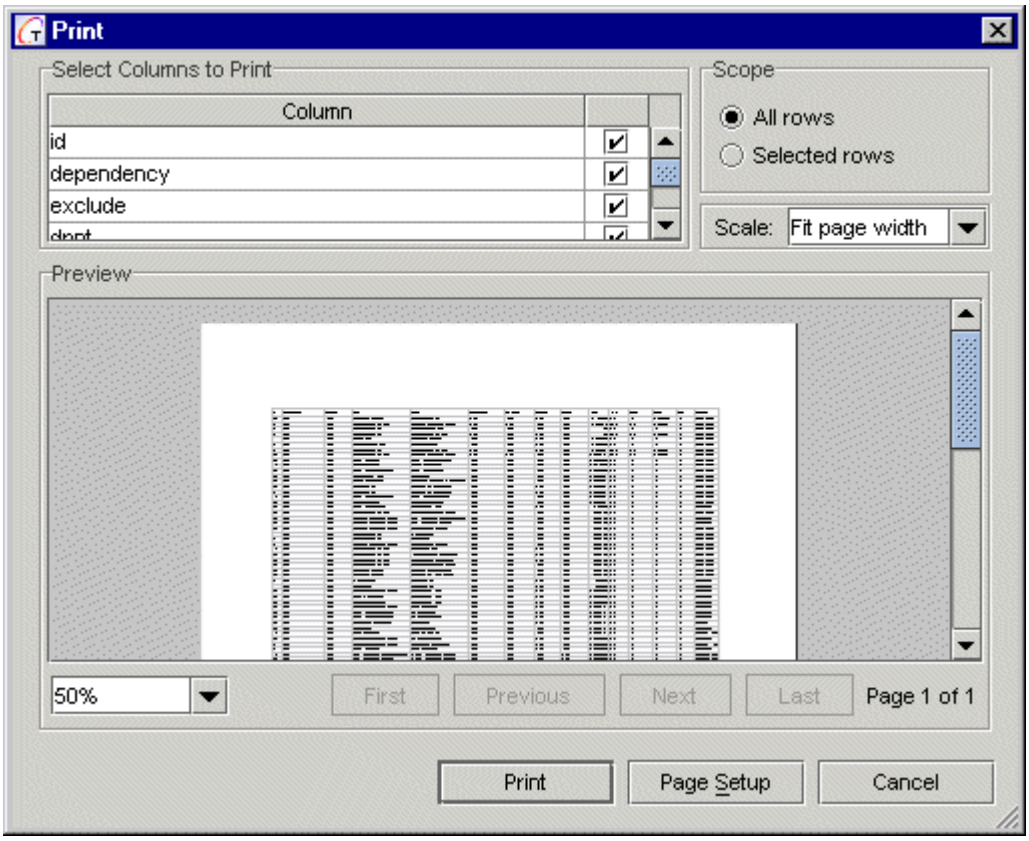


Abbildung 19: Dialog zum Drucken der Metrikergebnisse

Wählen Sie im linken oberen Teil des Dialogs diejenigen Spalten der Metriktabelle aus, die Sie auf Ihrem Ausdruck dargestellt haben möchten. Nachdem Sie ggf. weitere Angaben zu den Druckereinstellungen gemacht haben, betätigen Sie die Schaltfläche **Print** um den Ausdruck zu starten.

6.3 Graphanzeige

6.3.1 Speichern des Abhängigkeitsgraphen im XML-Format

Um den Abhängigkeitsgraphen zur weiteren Analyse im XML-Format zu speichern, wählen Sie das Untermenü **Export...** im Kontextmenü der Graphanzeige und dort den Menüpunkt **As XML...**. Hierauf öffnet sich folgender Dialog:

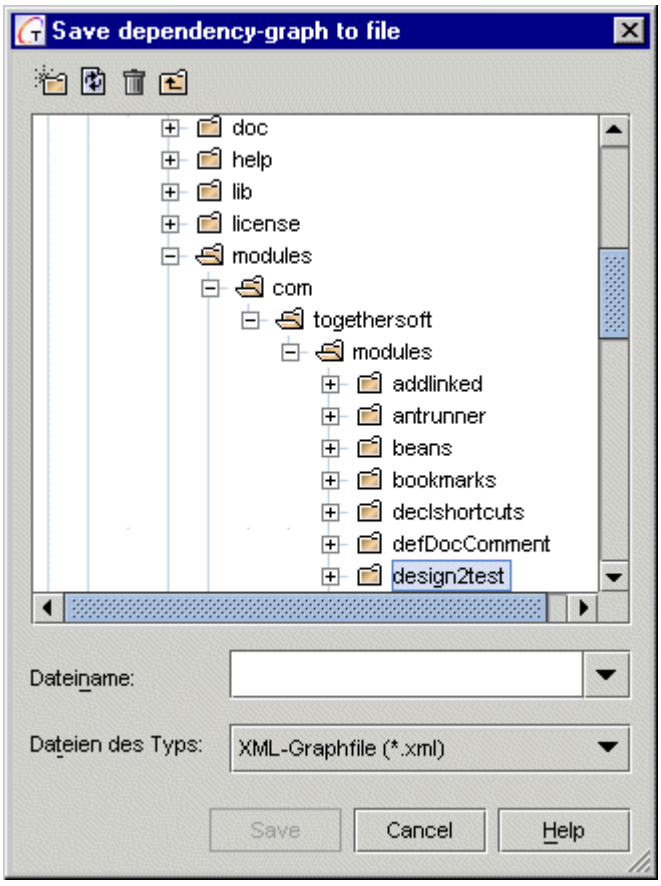


Abbildung 20: Dialog zum Speichern des Abhängigkeitsgraphen

Wählen Sie das Verzeichnis und geben Sie den Namen der XML-Datei an, unter dem der Abhängigkeitsgraph gespeichert werden soll. Um den Speichervorgang abzuschließen, drücken Sie die Schaltfläche **Save**.

6.3.2 Speichern des Abhängigkeitsgraphen im Grafikformat

Um den Abhängigkeitsgraphen im Grafikformat (JPEG) zu speichern, wählen Sie das Untermenü **Export...** im Kontextmenü der Graphanzeige und dort den Menüpunkt **As Image...**. Der daraufhin angezeigte Dialog gleicht dem zum Speichern im XML-Format (→ *Abbildung 20: Dialog zum Speichern des Abhängigkeitsgraphen*).

Wählen Sie das Verzeichnis und geben Sie den Namen der Grafikdatei an, unter dem das Bild des aktuell angezeigten Abhängigkeitsgraphen gespeichert werden soll. Um den Speichervorgang abzuschließen, drücken Sie die Schaltfläche **Save**.

6.3.3 Druck der Graphanzeige

ACHTUNG: Um die Druckmenüs von Design2Test nutzen zu können, muss das Quality Assurance (QA)-Modul von Together geladen sein (→ *1 Installation*).

Um die aktuelle Graphanzeige auszudrucken, wählen Sie den Menüpunkt **Print...** im Kontextmenü des Graphanzeige-Fensters. Hierauf öffnet sich folgender Druck-Dialog:

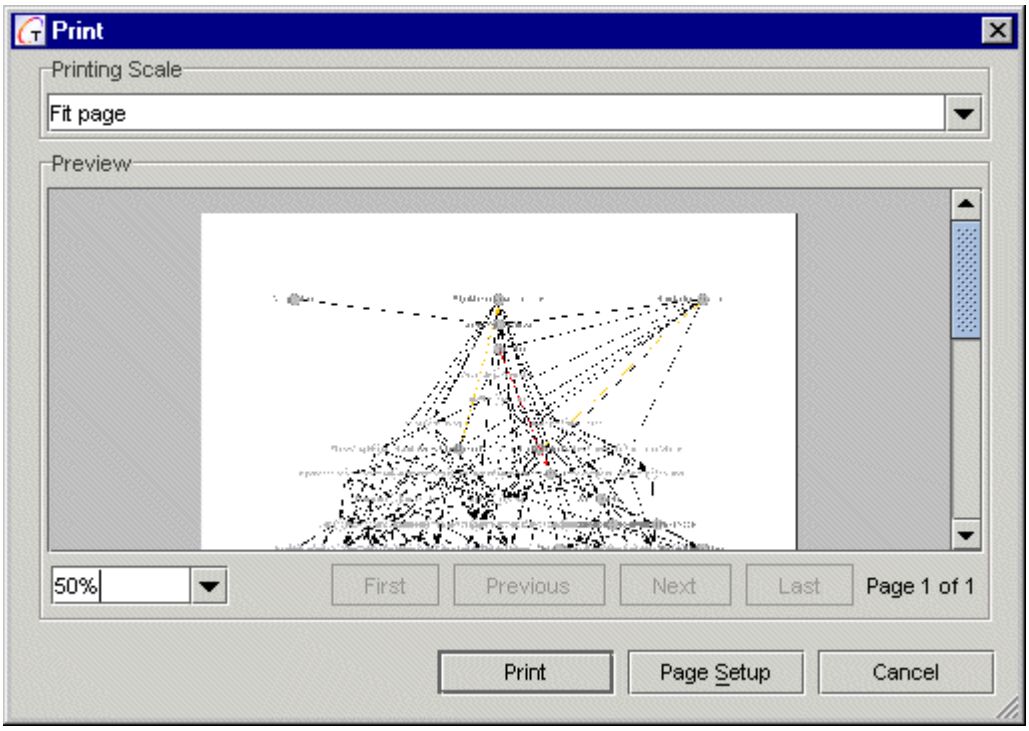


Abbildung 21: Dialog zum Drucken der aktuellen Graphanzeige

Wählen Sie die gewünschten Druckeinstellungen aus, und betätigen Sie anschließend die Schaltfläche **Print** um den Ausdruck zu starten.

Anhang Kontextmenüs - Kurzübersichten

A.1 Metriktabellen

Klicken Sie mit der rechten Maustaste in eine der beiden Metriktabellen, so erhalten Sie ein spaltenabhängiges Kontext-Menü:

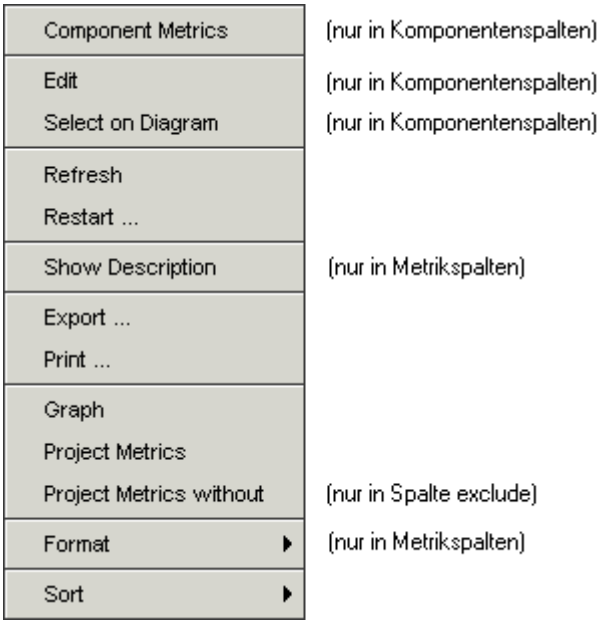


Abbildung 22: Kontextmenü bei der Anzeige der Abhängigkeitsmetriken

Die einzelnen Menüpunkte bieten Ihnen folgende Möglichkeiten.

- **Component Metrics:** Bringt das Unterpanel **Component Metrics** (→ *4.2 Komponentenmetriken*) in den Vordergrund und zeigt die zur angeklickten Klasse gehörenden Komponentenmetriken an.
- **Edit:** Öffnet die angeklickte Komponente im Texteditor von Together.
- **Select on Diagram:** Öffnet das Klassendiagramm, dessen Bestandteil die angeklickte Komponente ist, in der Designer Pane von Together. Die angeklickte Komponente wird innerhalb des Diagramms selektiert.
- **Refresh:** Startet für Abhängigkeits- *und* Komponentenmetriken eine neue Metrikberechnung mit den zuletzt ausgewählten Metriken und den zuletzt eingestellten Optionen.
- **Restart...:** Öffnet einen neuen Metrikdialog. Die Metrikauswahl für Abhängigkeits- *und* Komponentenmetriken kann neu vorgenommen werden, die Optionen können neu eingestellt werden.
- **Show Description:** Zeigt eine kurze Erläuterung der angeklickten Metrik an.
- **Export...:** Möglichkeiten zum Speichern der Metriktabelle zur weiteren Verarbeitung im Textformat.
- **Print...:** Möglichkeit zum Ausdrucken der Ergebnisse der Metriktabelle.
- **Graph:** Graphische Anzeige des Abhängigkeitsgraphen unter Berücksichtigung der errechneten Metriken (→ *5 Graphische Anzeige der Abhängigkeiten*)
- **Project Metrics:** Öffnet ein kleines Fenster, in dem die Metrikwerte des Gesamtprojektes angezeigt werden (→ *4.4.1 Projektmetriken für das Gesamtprojekt*).

- **Project Metrics without:** Öffnet ein kleines Fenster, in dem die Metrikwerte ohne die in der Spalte **exclude** abgewählten Abhängigkeiten angezeigt werden (→ *4.4.2 Projektmetriken nach Beseitigung von Abhängigkeiten*).
- **Format | Absolute Values:** Stellt die Werte einer nummerischen Metrik in absoluten Werten dar.
- **Format | Values as Percentage:** Stellt die Werte einer nummerischen Metrik in Prozentform dar. Hierbei handelt es sich um den Grad der Verringerung des korrespondierenden Metrikwertes des Gesamtprojektes nach Entfernung der Abhängigkeit.
- **Sort | Ascending/Descending:** Sortiert nach der jeweiligen Spalte in der angegebenen Ordnung.

Eine detaillierte Beschreibung der einzelnen Funktionen finden Sie in Abschnitt *4.3 Funktionen in den Metrikanzeigen.*

A.2 Komponententabellen

Klicken Sie mit der rechten Maustaste in eine der beiden Übersichten über verbundene Komponenten im rechten Teil des Panels **Component Metrics**, so steht Ihnen folgendes Kontextmenü zur Verfügung:

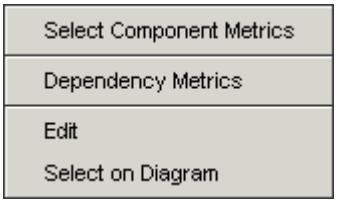


Abbildung 23: Kontextmenü bei der Anzeige verbundener Komponenten

Die einzelnen Menüpunkte bieten Ihnen folgende Möglichkeiten:

- **Select Component Metrics:** Wechselt in der Metriktabelle zu der zur ausgewählten Komponente gehörende Zeile mit den Metrikwerten. Gleichzeitig erfolgt auch die Aktualisierung der geöffneten Komponentenfenster im rechten Teil der Anzeige. Auf diese Weise können Sie in beliebiger Richtung entlang eines Pfads im Abhängigkeitsgraphen navigieren.
- **Dependency Metrics:** Aktiviert das Unterpanel **Dependency Metrics** und zeigt dort die Metrikwerte der zugrunde liegenden Abhängigkeit an.
- **Edit:** Öffnet die angeklickte Komponente im Texteditor von Together.
- **Select on Diagram:** Öffnet das Klassendiagramm, dessen Bestandteil die angeklickte Komponente ist, in der Designer Pane von Together. Die angeklickte Komponente wird innerhalb des Diagramms selektiert.

Eine detaillierte Beschreibung der einzelnen Funktionen finden Sie in Abschnitt *4.3 Funktionen in den Metrikanzeigen.*

A.3 Graphanzeige

Wenn Sie mit rechten Maustaste in die Graphanzeige klicken, so erhalten Sie ein vom ausgewählten Graphelement abhängiges Kontextmenü:

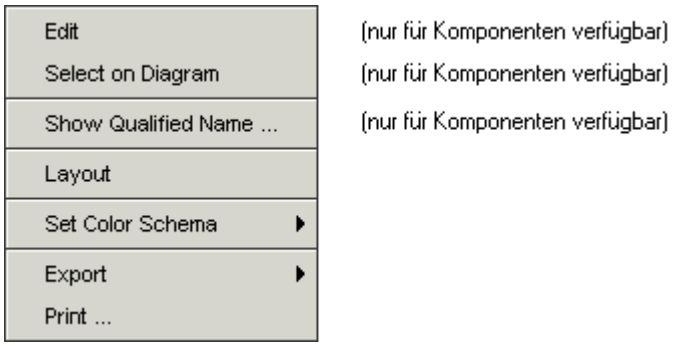


Abbildung 24: Kontextmenüs in der Graphanzeige

Hinter den angebotenen Menüpunkten verbergen sich im Einzelnen folgende Funktionalitäten:

- **Edit:** Öffnet die angeklickte Komponente im Texteditor von Together.
- **Select on Diagram:** Öffnet das Klassendiagramm, dessen Bestandteil die angeklickte Komponente ist, in der *Designer Pane* von Together. Die angeklickte Komponente wird innerhalb des Diagramms selektiert.
- **Show Qualified Name...:** Zeigt den vollständig qualifizierten Namen der ausgewählten Komponente an.
- **Layout**: Ordnet die angezeigten Komponenten und Abhängigkeiten neu und aktualisiert die Anzeige.
- **Set Color Schema | Normal**: Schaltet zum Standard-Farbschema um.
- **Set Color Schema | Show Component**: Installiert ein Farbschema zur Anzeige von Zyklen.
- **Set Color Schema | Show Dependency Type**: Installiert ein Farbschema zur Anzeige der Art der Abhängigkeit.
- **Export | As Image...:** Möglichkeiten zum Speichern des Graphen zur weiteren Bearbeitung in einem Grafikformat (JPEG).
- **Export | As XML...:** Möglichkeiten zum Speichern des Graphen zur weiteren Bearbeitung im XML-Format.
- **Print...:** Möglichkeit zum Ausdrucken der aktuellen Graphanzeige.

Eine detaillierte Beschreibung der einzelnen Funktionen finden Sie in Abschnitt *5.2 Funktionen in der Graphanzeige.*